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

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

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

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

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).

Organization of this book

This book is divided into four sections. Part I, "Kinect at a glance," provides a quick tour of Kinect for Windows SDK and describes how it works. Part II, "Integrate Kinect in your application," shows you how to develop tools to integrate Kinect seamlessly into your own applications. Part III, "Postures and gestures," focuses on how to develop a rich postures and gestures recognition system. Finally, Part IV, "Creating a user interface for Kinect," covers the use of Kinect as a new input mechanism and describes how you can create an augmented reality application.

Finding your best starting point in this book

The different sections cover a wide range of technologies associated with the Kinect for Windows SDK. Depending on your needs and your familiarity with Kinect, you may want to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Start reading
New to Kinect for Windows development	Chapter 1
Familiar with Kinect and interested in gestures and postures development	Chapter 5

Most of the book's chapters include hands-on samples that let you try out the concepts just learned. No matter which sections you choose to focus on, be sure to download and install the sample applications on your system.

System requirements

You will need the following hardware and software to use the code presented in this book:

- Windows 7 or Windows 8 (32-bit or 64-bit edition)
- Microsoft Visual Studio 2010 Express or other Visual Studio 2010 edition
- .NET Framework 4 (installed with Visual Studio 2010)
- XNA Game Studio 4
- 32-bit (x86) or 64-bit (x64) processors
- Dual-core, 2.66-GHz or faster processor
- USB 2.0 bus dedicated to the Kinect

- 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 at oreilly.com:

http://go.microsoft.com/FWLink/?Linkid=258659

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*.

PART I

Kinect at a glance

CHAPTER 1

A bit of background

The development of a motion-sensing input device by Microsoft was first announced under the code name Project Natal on June 1, 2009, at the E3 2009. Kinect was launched in November 2010 and quickly became the fastest selling consumer electronics device according to Guinness World Records.

On June 16, 2011, Microsoft announced the release of the Kinect for Windows Software Development Kit (SDK). Early in 2012, Microsoft shipped the commercial version of the SDK, allowing developers all around the world to use the power of the Kinect sensor in their own applications. A few months later, in June 2012, Microsoft shipped Kinect SDK 1.5, and that version is used in this book.

The sensor

Think of the Kinect sensor as a 3D camera—that is, it captures a stream of colored pixels with data about the depth of each pixel. It also contains a microphone array that allows you to capture positioned sounds. The Kinect sensor's ability to record 3D information is amazing, but as you will discover in this book, it is much more than just a 3D camera.

From an electronics point of view, the Kinect sensor uses the following equipment:

- A microphone array
- An infrared emitter
- An infrared receiver
- A color camera

The sensor communicates with the PC via a standard USB 2.0 port, but it needs an additional power supply because the USB port cannot directly support the sensor's power consumption. Be aware of this when you buy a Kinect sensor—you must buy one with a special USB/power cable. For more information, see *http://www.microsoft.com/en-us/kinectforwindows/purchase/*.

Figure 1-1 shows the internal architecture of a sensor.

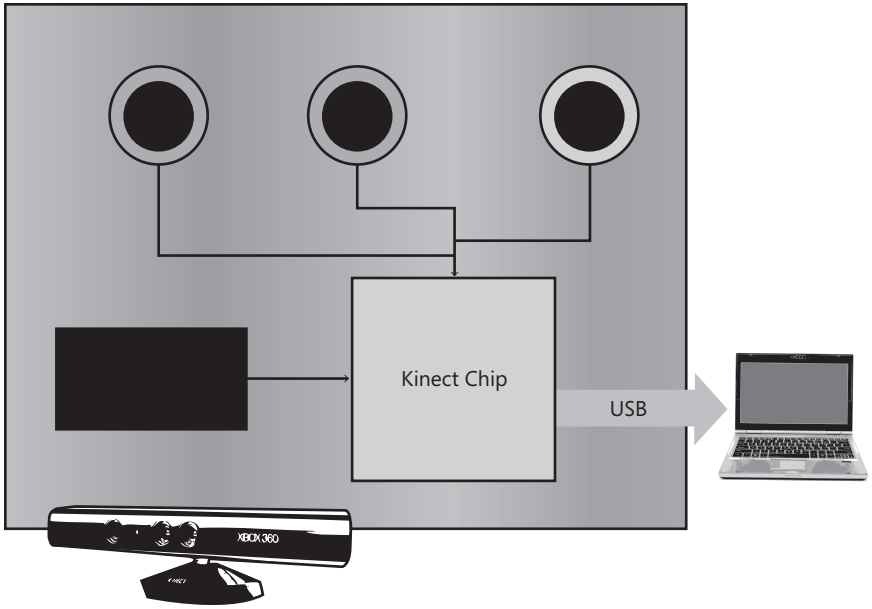

FIGURE 1-1 The inner architecture of a Kinect sensor.

Because there is no CPU inside the sensor—only a digital signal processor (DSP), which is used to process the signal of the microphone array—the data processing is executed on the PC side by the Kinect driver.

The driver can be installed on Windows 7 or Windows 8 and runs on a 32- or 64-bit processor. You will need a least 2 GB of RAM and a dual-core 2.66-GHz or faster processor for the Kinect driver.

One more important point is that you will not be able to use your Kinect under a virtual machine because the drivers do not yet support a virtualized sandbox environment.

Limits

The sensor is based on optical lenses and has some limitations, but it works well under the following ranges (all starting from the center of the Kinect):

- Horizontal viewing angle: 57°
- Vertical viewing angle: 43°
- User distance for best results: 1.2m (down to 0.4m in near mode) to 4m (down to 3m in near mode)
- Depth range: 400mm (in near mode) to 8000mm (in standard mode)
- Temperature: 5 to 35 degrees Celsius (41 to 95 degrees Fahrenheit)

The vertical viewing position can be controlled by an internal servomotor that can be oriented from –28° to +28°. Figure 1-2 illustrates the limits of the Kinect sensor to achieve best results.

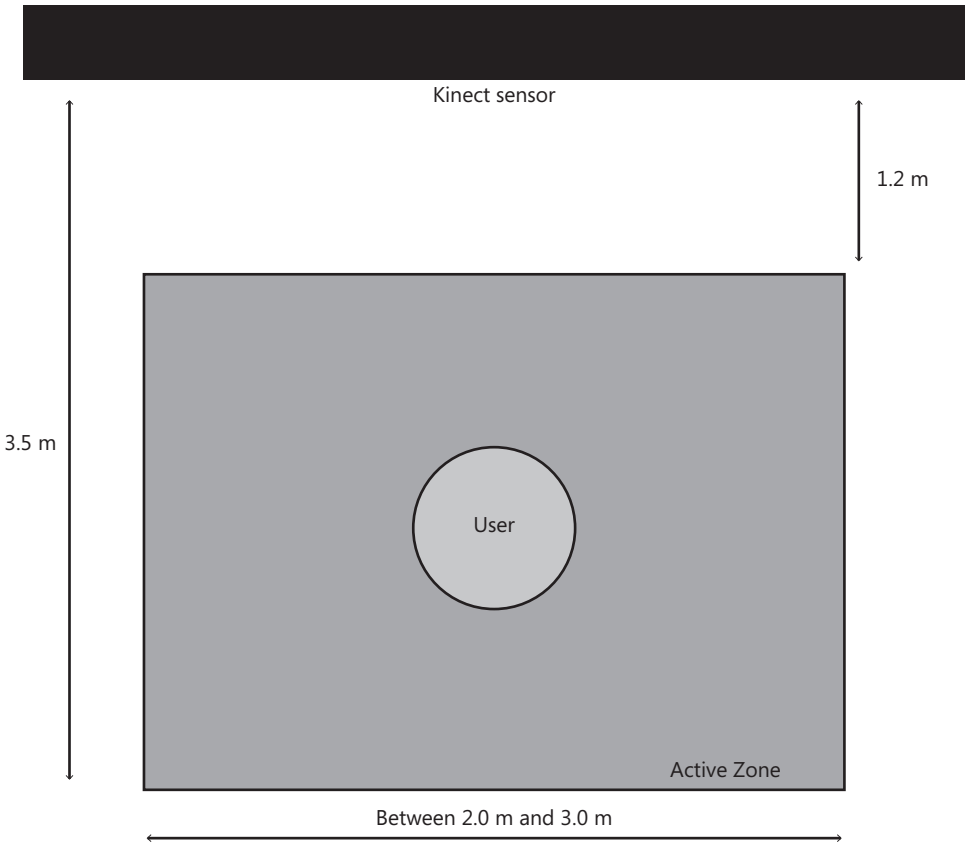

FIGURE 1-2 Limits of the Kinect sensor in standard mode for accurate results.

Furthermore, follow these guidelines for a successful Kinect experience:

- Do not place the sensor on or in front of a speaker or on a surface that vibrates or makes noise.
- Keep the sensor out of direct sunlight.
- Do not use the sensor near any heat sources.

A specific mode called *near mode* allows you to reduce the detection distance.

The Kinect for Windows SDK

To develop an application using a Kinect sensor, you must install the Kinect for Windows SDK, available at *http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx*. Following are the requirements for using the SDK:

- Microsoft Visual Studio 2010 Express or other Visual Studio 2010 edition
- .NET Framework 4 (installed with Visual Studio 2010)

The SDK is available for C++ and managed development. The examples in this book use C# as the main development language. All the features described using C# are available in C++.

Starting with SDK 1.5, Microsoft introduced the Kinect for Windows Developer Toolkit, which contains source code samples and other resources to simplify development of applications using the Kinect for Windows SDK. You should download it as a companion to the SDK.

Using a Kinect for Xbox 360 sensor with a developer computer

It is possible to develop applications on your developer computer using a Kinect for Xbox 360 sensor. If you do so, the SDK will produce a warning similar to this one:

> *The Kinect plugged into your computer is for use on the Xbox 360. You may continue using your Kinect for Xbox 360 on your computer for development purposes. Microsoft does not guarantee full compatibility for Kinect for Windows applications and the Kinect for Xbox 360.*

As you can see, you can use your Xbox 360 sensor with a computer, but there is no guarantee that there will be full compatibility between the sensor and your computer in the future.

Furthermore, on an end user computer (one on which the SDK is not installed), your initialization code will fail, prompting a "Device not supported" message if you run it with a Kinect for Xbox 360 sensor.

Preparing a new project with C++

To start a new project with C++, you must include one of the following headers:

- **NuiApi.h** Aggregates all NUI API headers and defines initialization and access functions for enumerating and accessing devices
- **NuiImageCamera.h** Defines the APIs for image and camera services so you can adjust camera settings and open streams and read image frames
- **NuiSkeleton.h** Defines the APIs for skeleton data so you can get and transform skeleton data
- **NuiSensor.h** Defines the Audio API that returns the audio beam direction and source location

These headers are located in the Kinect SDK folder: <Program Files>\Microsoft SDKs\Kinect\vX.XX\inc. The library MSRKinectNUI.lib is located in <Program Files>\Microsoft SDKs\Kinect\vX.XX\lib.

Preparing a new project with C#

To start a new project with Kinect for Windows SDK in C#, you must choose between Windows Forms and WPF. The examples in this book use WPF, as shown in Figure 1-3.

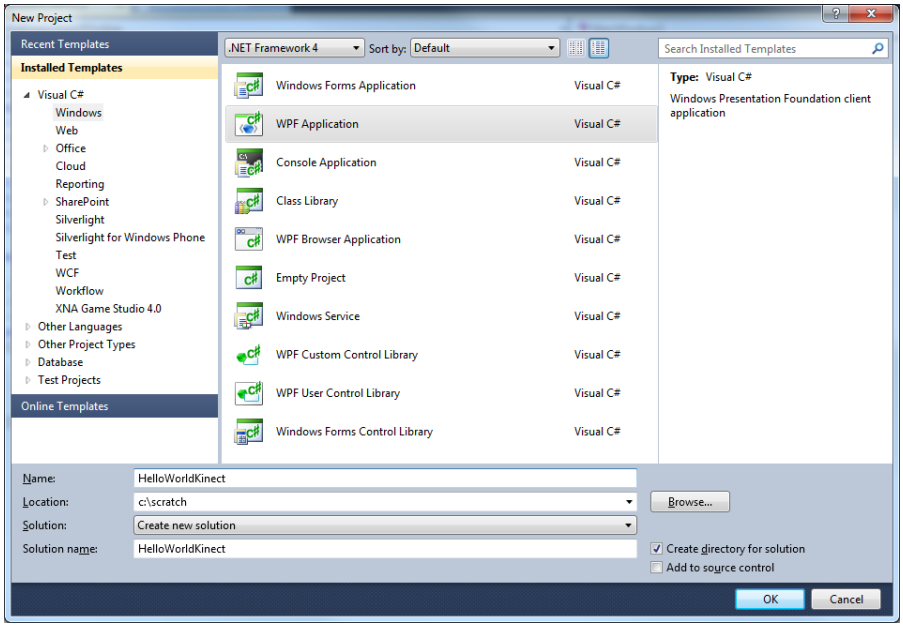

FIGURE 1-3 The Visual Studio projects list.

After you have created your project, you can add a reference to the Kinect for Windows assembly, as shown in Figure 1-4.

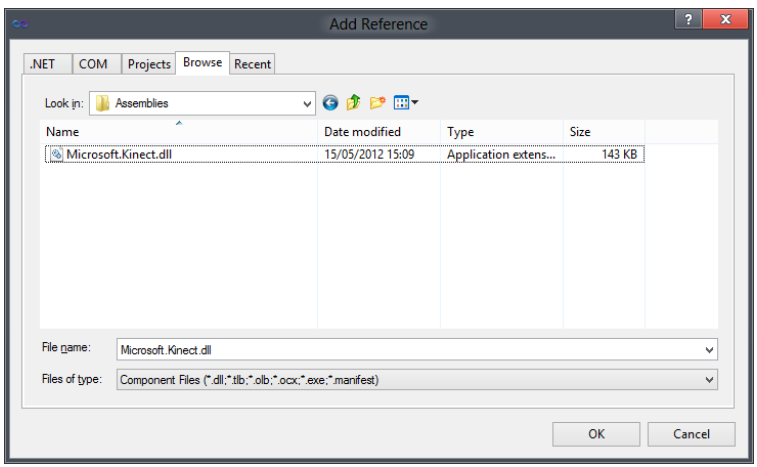

FIGURE 1-4 Selecting Kinect assembly.

And you're done!

Using the Kinect for Windows SDK

For your first application, you will simply set up the initialization and cleaning functionality. Kinect for Windows SDK gracefully offers you a way to detect current connected devices and will raise an event when anything related to sensors changes on the system.

Following is the code you must implement in the main window (the *Kinects_StatusChanged* and *Initialize* methods are explained following the code):

```
using System;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Kinect;

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    KinectSensor kinectSensor;
    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        try
        {
            //listen to any status change for Kinects
            KinectSensor.KinectSensors.StatusChanged += Kinects_StatusChanged;

            //loop through all the Kinects attached to this PC, and start the first that is
connected without an error.
            foreach (KinectSensor kinect in KinectSensor.KinectSensors)
            {
                if (kinect.Status == KinectStatus.Connected)
                {
                    kinectSensor = kinect;
                    break;
                }
            }

            if (KinectSensor.KinectSensors.Count == 0)
                MessageBox.Show("No Kinect found");
            else
                Initialize();   // Initialization of the current sensor

        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
}
```

As you can see, you check to see if there is already a device connected that needs to initialize, or you can wait for a device to connect using the following event handler:

```
void Kinects_StatusChanged(object sender, StatusChangedEventArgs e)
{
    switch (e.Status)
    {
        case KinectStatus.Connected:
            if (kinectSensor == null)
            {
                kinectSensor = e.Sensor;
                Initialize();
            }
            break;
        case KinectStatus.Disconnected:
            if (kinectSensor == e.Sensor)
            {
                Clean();
                MessageBox.Show("Kinect was disconnected");
            }
            break;
        case KinectStatus.NotReady:
            break;
        case KinectStatus.NotPowered:
            if (kinectSensor == e.Sensor)
            {
                Clean();
                MessageBox.Show("Kinect is no longer powered");
            }
            break;
        default:
            MessageBox.Show("Unhandled Status: " + e.Status);
            break;
    }
}
```

Based on the current status (Connected, Disconnected, NotReady, NotPowered), you will call *Clean()* to dispose of previously created resources and events, and you will call *Initialize()* to start a new sensor:

```
private void Initialize()
{
    if (kinectSensor == null)
        return;
    kinectSensor.Start();
}

private void Clean()
{
    if (kinectSensor != null)
    {
        kinectSensor.Stop();
        kinectSensor = null;
    }
}
```

CHAPTER 2

Who's there?

The Kinect sensor can be defined as a multistream source—it can provide a stream for every kind of data it collects. Because the Kinect sensor is a color/depth/audio sensor, you can expect it to send three different streams. In this chapter, you'll learn about the kinds of data provided by these streams and how you can use them. Furthermore, you'll see how the Kinect for Windows SDK can compute a complete tracking of skeletons detected in front of the sensor using only these raw streams.

SDK architecture

Before you learn more about streams, you need to become familiar with the inner structure of the Kinect for Windows SDK, shown in Figure 2-1. The SDK installs the natural user interface (NUI) application programming interface (API) and the Microsoft Kinect drivers to integrate the Kinect sensor within Microsoft Windows.

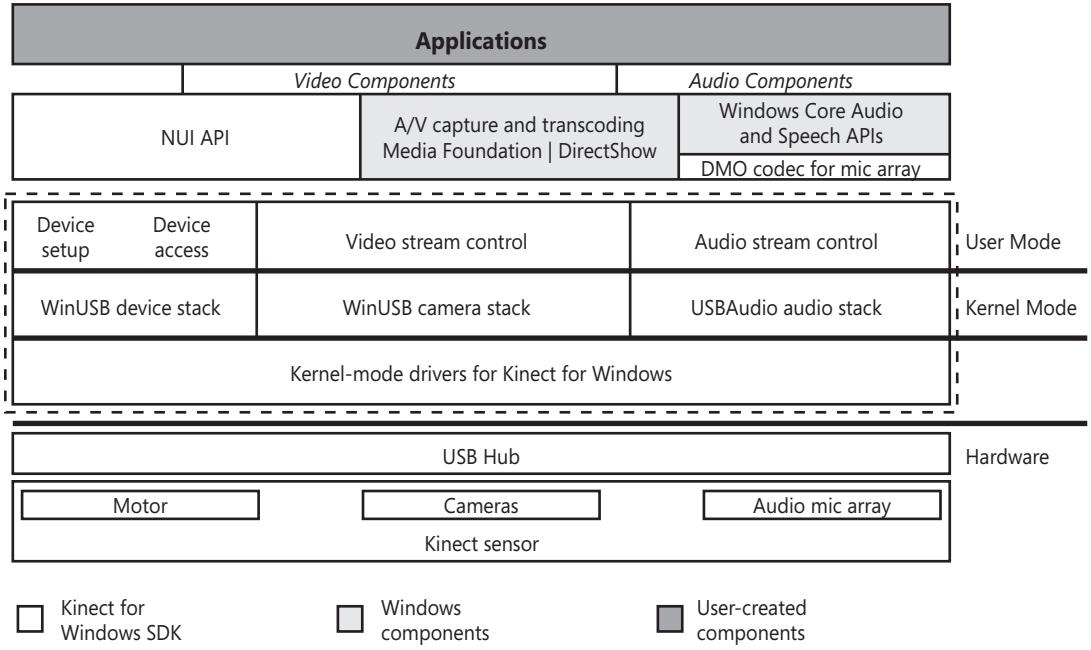

FIGURE 2-1 The Kinect for Windows SDK architecture.

Note The Kinect for Windows SDK is compatible with both Windows 7 and Windows 8 and with x86 and x64 architectures.

As you can see, the streams are transmitted to the PC using a USB hub. The NUI API collects raw data and presents it to applications. But the SDK also integrates in Windows through standard components including

- Audio, speech, and media API to use with applications such as the Microsoft Speech SDK.
- DirectX Media Object (DMO) to use with applications such as DirectShow or Media Foundation.

Let's have a look at each stream and its related data provided by the Kinect for Windows SDK.

The video stream

Obviously, the first data provided by the Kinect sensor is the video stream. Although it functions as a 3D camera, at its most basic level, Kinect is a standard camera that can capture video streams using the following resolutions and frame rates:

- 640 × 480 at 30 frames per second (FPS) using red, green, and blue (RGB) format
- 1280 × 960 at 12 FPS using RGB format
- 640 × 480 at 15 FPS using YUV (or raw YUV) format

The RGB format is a 32-bit format that uses a linear X8R8G8B8 formatted color in a standard RGB color space. (Each component can vary from 0 to 255, inclusively.)

The YUV format is a 16-bit, gamma-corrected linear UYUY-formatted color bitmap. Using the YUV format is efficient because it uses only 16 bits per pixel, whereas RGB uses 32 bits per pixel, so the driver needs to allocate less buffer memory.

Because the sensor uses a USB connection to pass data to the PC, the bandwidth is limited. The Bayer color image data that the sensor returns at 1280 × 1024 is compressed and converted to RGB before transmission to the Kinect runtime. The runtime then decompresses the data before it passes the data to your application. The use of compression makes it possible to return color data at frame rates as high as 30 FPS, but the algorithm used for higher FPS rates leads to some loss of image fidelity.

Using the video stream

The video data can be used to give visual feedback to users so that they can see themselves interacting with the application, as shown in Figure 2-2. For example, you can add the detected skeleton or other information on top of the video to create an augmented reality experience, as you will see in Chapter 11, "Creating augmented reality with Kinect."

FIGURE 2-2 Using Kinect video frame to produce augmented reality applications.

In terms of code, it's simple to activate the video stream (called the "color stream" by the SDK):

```
var kinectSensor = KinectSensor.KinectSensors[0]; KinectSensor.KinectSensorskinectSensor.
ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
kinectSensor.Start();
```

With *KinectSensor.ColorStream.Enable()* we can choose the requested format and frame rate by using the following enumeration:

- *RgbResolution640×480Fps30* (default)
- *RgbResolution1280×960Fps12*
- *RawYuvResolution640×480Fps15*
- *YuvResolution640×480Fps15*

Getting frames

You can use two different approaches to get video data. First, you can use the event approach and register an event handler:

```
kinectSensor.ColorFrameReady += kinectSensor_ColorFrameReady;
void kinectSensor_ColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)
{
}
```

You can also poll for a new frame by using the following code:

```
ImageFrame frame = kinectSensor.ColorStream.OpenNextFrame(500);
```

In this case, request the next frame specifying the timeout to use.

> **Note** In a general way, each stream can be accessed through an event or through direct request (polling). But if you choose one model to use, you won't be able to change to a different model later.

Stream data is a succession of static images. The runtime continuously fills the frame buffer. If the application doesn't request the frame, the frame is dropped and the buffer is reused.

In polling mode, the application can poll for up to four frames in advance. It is the responsibility of the application to define the most adequate count. In Chapter 3, "Displaying Kinect data," you'll see how you can display and use this color data.

And, of course, you have to close the *ColorStream* using the *Disable()* method:

```
kinectSensor.ColorStream.Disable();
```

Although it is not mandatory, it's a good habit to clean and close the resources you use in your application.

The depth stream

As we just saw, the Kinect sensor is a color camera—but it is also a depth camera. Indeed, the sensor can send a stream composed of the distance between the camera plane and the nearest object found. Each pixel of the resulting image contains the given distance expressed in millimeters.

Figure 2-3 shows a standard depth stream display with player identification.

Using the depth stream

To initialize and use the depth stream, you use code similar to what you use for the video stream:

```
var kinectSensor = KinectSensor.KinectSensors[0];
kinectSensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
```

The following resolutions are supported:

- 640 × 480
- 320 × 240
- 80 × 60

All resolutions use a frame rate of 30 FPS. Using the *Enable* method of the *DepthStream*, you can select the resolution you prefer with the following enumeration:

- *Resolution640x480Fps30*
- *Resolution320x240Fps30*
- *Resolution80x60Fps30*

FIGURE 2-3 Depth value samples with player identification.

Getting frames

Again, as with the video stream, you can use an event model or a polling model. The polling model is available through the *OpenNextFrame* method:

```
var frame = kinectSensor.DepthStream.OpenNextFrame(500);
```

The event model will use the following signature:

```
kinectSensor.DepthFrameReady += kinectSensor_DepthFrameReady;
void kinectSensor_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)
{
}
```

An application must choose one model or the other; it cannot use both models simultaneously.

Computing depth data

Figure 2-4 shows how depth values are computed.

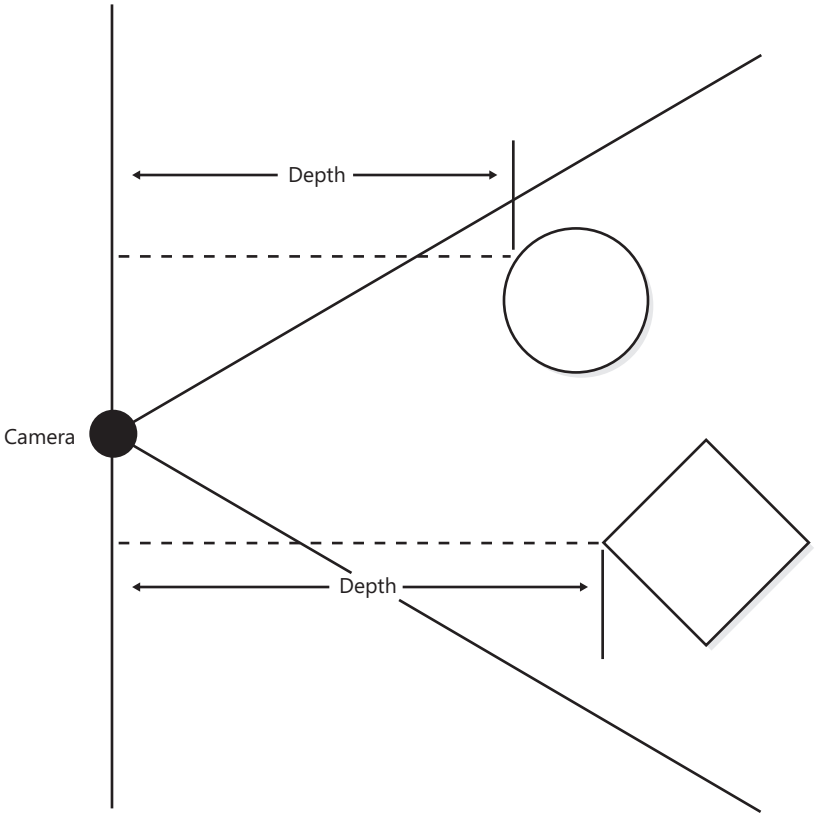

FIGURE 2-4 Evaluation of depth values.

There are two possible range modes for the depth values. We can use near or standard mode. Using standard mode, the values are between 800mm and 4000mm inclusively; using near mode, the values are between 400mm and 3000mm.

In addition, the SDK will return specific values for out-of-range values:

- Values under 400mm or beyond 8000mm are marked as [Unknown].
- In near mode, values between 3000mm and 8000mm are marked as [Too far].
- In standard mode, values between 4000mm and 8000mm are marked as [Too far] and values 400mm and 800mm are marked as [Too near].

To select the range mode, execute the following code:

```
var kinectSensor = KinectSensor.KinectSensors[0];
kinectSensor.DepthStream.Range = DepthRange.Near;
```

The depth stream stores data using 16-bit values. The 13 high-order bits of each pixel contain the effective distance between the camera plane and the nearest object in millimeters. The three low-order bits of each pixel contain the representation of the player segmentation map of the current pixel.

The player segmentation map is built by the Kinect system—when skeleton tracking is activated—and is a bitmap in which each pixel corresponds to the player index of the closest person in the field of view of the camera. A value of zero indicates that there is no player detected. The three low-order bits must be treated as an integer value.

Using all this information, you can easily produce a picture like the one you saw in Figure 2-3, in which red pixels (which appear in the printed book as the dark gray area on top of the body) indicate a pixel with a player index that is not zero.

The following code demonstrates how to get the distance and the player index of each pixel (considering that *depthFrame16* contains the pixels list in a form of *ushort[]* and *i16* is the current pixel index):

```
int user = depthFrame16[i16] & 0x07;
int realDepth = (depthFrame16[i16] >> 3);
```

As you can see, you must use some bitwise operations to get the information you need from each pixel. The user index is on the three low-order bits, so a simple mask with 00000111 in binary form or 0×07 in hexadecimal form can extract the value. To get the depth value, you can remove the first three bits by offsetting the pixels to the right with the >> operator.

A depth value of 0 indicates that no depth data is available at the given coordinate (x, y) because all the objects were either too far from or too close to the camera.

Chapter 3 describes how to display and use this depth data.

The audio stream

The Kinect sensor features a microphone array that consists of four microphones several centimeters apart and arranged in a linear pattern. This structure allows really interesting functionalities:

- Effective noise suppression.
- Acoustic echo cancellation (AEC).
- Beamforming and source localization. Each microphone in the array will receive a specific sound at a slightly different time, so it's possible to determine the direction of the audio source. You can also use the microphone array as a steerable directional microphone.

The Kinect for Windows SDK allows you to

- Capture high-quality audio.
- Use the KinectAudio DMO built-in algorithms to control the sound "beam" and provide the source direction to your code.
- Use the Microsoft.Speech speech recognition API.

The following code provides access to the audio stream:

```
var kinectSensor = KinectSensor.KinectSensors[0];
KinectAudioSource source = kinectSensor.AudioSource;
using (Stream sourceStream = source.Start())
{
}
```

The returned stream is encoded using a 16 KHz, 16-bit PCM format.

After *Start* is called, the *KinectAudioSource.SoundSourcePosition* property is updated continuously and will contain the audio source direction. This value is an angle (in radians) to the current position of the audio source. The angle is relative to the z axis of the camera, which is perpendicular to the Kinect sensor.

This value has a confidence level associated on the *KinectAudioSource.SoundSourcePositionConfidence* property. This property is updated only when *BeamAngleMode* is set to *BeamAngleMode.Automatic* or *BeamAngleMode.Adaptive*.

The *KinectAudioSource* class also has a lot of properties to control the audio stream. (Don't forget to set FeatureMode to true if you want to override default values.)

- ***AcousticEchoSuppression*** Gets or sets the number of times the system performs acoustic echo suppression.
- ***BeamAngle*** Specifies which beam to use for microphone array processing. The center value is zero, negative values refer to the beam to the right of the Kinect device (to the left of the user), and positive values indicate the beam to the left of the Kinect device (to the right of the user).
- ***BeamAngleMode*** Defines the current mode of the microphone array. Values can be
  - ***Automatic*** Perform beamforming. The system selects the beam.
  - ***Adaptative*** Perform adaptive beamforming. An internal source localizer selects the beam.
  - ***Manual*** Perform beamforming. The application selects the beam.
- ***ManualBeamAngle*** Beam angle to use when *BeamAngleMode* is set to manual.
- ***MaxBeamAngle*** The maximum beam angle (in radians). The center value is zero, negative values refer to the beam to the right of the Kinect device (to the left of the user), and positive values indicate the beam to the left of the Kinect device (to the right of the user).
- ***MinBeamAngle*** The minimum beam angle (in radians). The center value is zero, negative values refer to the beam to the right of the Kinect device (to the left of the user), and positive values indicate the beam to the left of the Kinect device (to the right of the user).
- ***EchoCancellationMode*** Defines the current echo cancellation mode. Values can be
  - ***CancellationAndSuppression*** Perform echo cancellation and suppression.
  - ***CancellationOnly*** Perform echo cancellation but not suppression.
  - ***None*** Do not perform echo cancellation or suppression.

- ***EchoCancellationSpeakerIndex*** Defines the index of the speaker to use for echo cancellation.
- ***NoiseSuppression*** Specifies whether to perform noise suppression. Noise suppression is a digital signal processing (DSP) component that suppresses or reduces stationary background noise in the audio signal. Noise suppression is applied after the AEC and microphone array processing.
- ***SoundSourceAngle*** The angle (in radians) to the current position of the audio source in camera coordinates, where the x and z axes define the horizontal plane. The angle is relative to the z axis, which is perpendicular to the Kinect sensor. After the *Start* method is called, this property is updated continuously.
- ***SoundSourceAngleConfidence*** The confidence associated with the audio source location estimate, ranging from 0.0 (no confidence) to 1.0 (total confidence). The estimate is represented by the value of the *SoundSourceAngle* property.
- ***MaxSoundSourceAngle*** The maximum sound source angle (in radians). The center value is zero, negative values refer to the beam to the right of the Kinect device (to the left of the user), and positive values indicate the beam to the left of the Kinect device (to the right of the user).
- ***MinSoundSourceAngle*** The minimum sound source angle (in radians). The center value is zero, negative values refer to the beam to the right of the Kinect device (to the left of the user), and positive values indicate the beam to the left of the Kinect device (to the right of the user).

As you can see, beamforming, audio quality, and processing control are quite simple with the Kinect for Windows SDK.

In Chapter 4, "Recording and playing a Kinect session," you'll learn how to use the audio stream to control your application.

Skeleton tracking

The NUI API uses the depth stream to detect the presence of humans in front of the sensor. Skeletal tracking is optimized to recognize users facing the Kinect, so sideways poses provide some challenges because parts of the body are not visible to the sensor.

As many as six people can be detected, and one or two people can be tracked at one time with the Kinect sensor. For each tracked person, the NUI will produce a complete set of positioned key points called a skeleton. A skeleton contains 20 positions, one for each "joint" of human body, as shown in Figure 2-5.

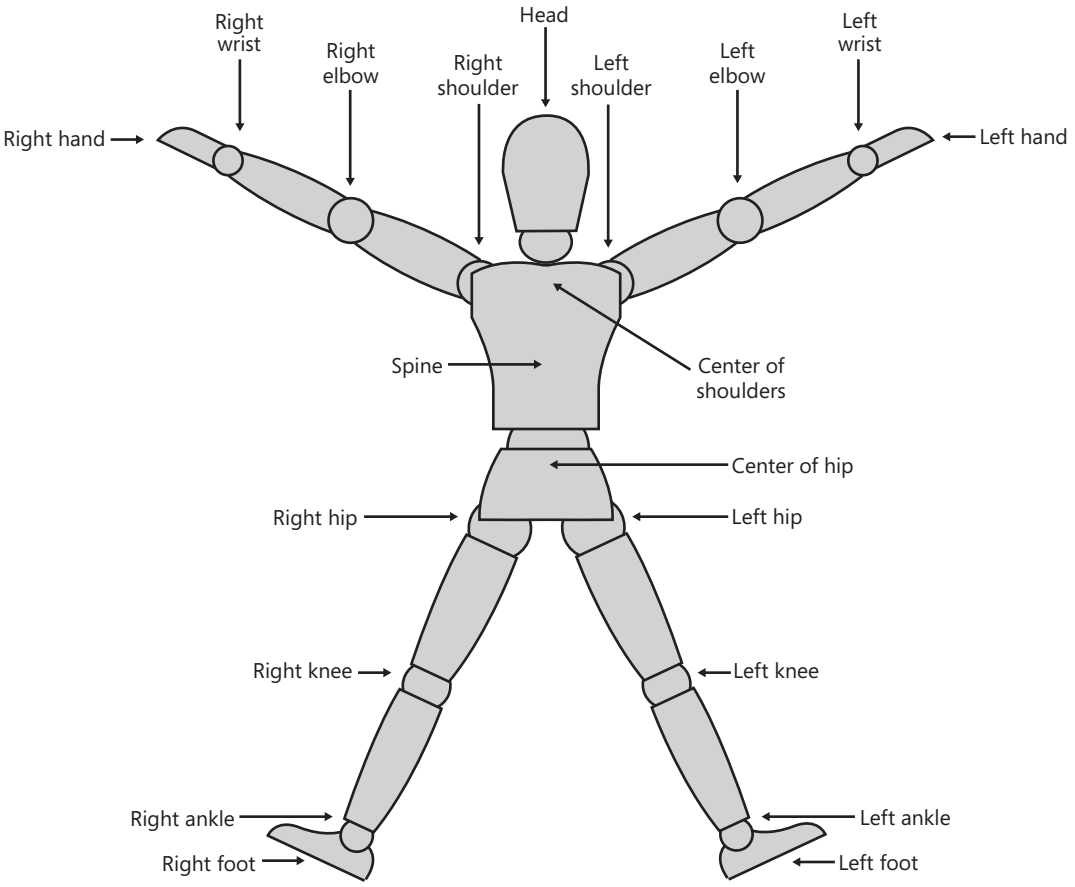

FIGURE 2-5 The 20 control points of a skeleton.

Each control point is defined by a position (x, y, z) expressed in skeleton space. The "skeleton space" is defined around the sensor, which is located at (0, 0, 0)—the point where the x, y, and z axes meet in Figure 2-6. Coordinates are expressed using meters (instead of millimeters, which are used for depth values).

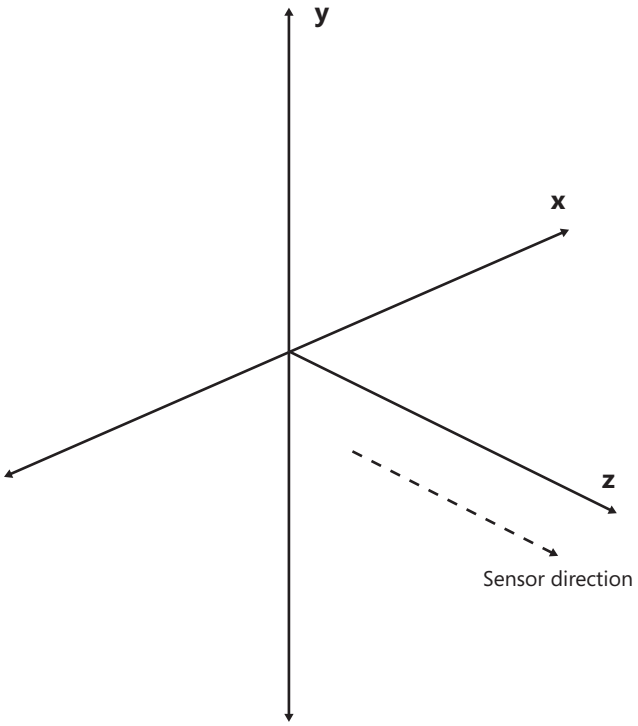

FIGURE 2-6 Skeleton space axes.

The x axis extends to the right (from the point of view of the user), the y axis extends upward, and the z axis is oriented from the sensor to the user.

Starting with SDK 1.5, Kinect for Windows SDK is now able to track sitting users. In this case only the 10 joints of the upper body (head, shoulders, elbows, arms, and wrists) are tracked. The lower body joints are reported as *NotTracked*.

Be aware that in seated mode, you should move your body slightly at startup to allow the system to recognize you. In default mode (standing up), the system uses the distance from the subject to the background to detect users. In seated mode, it uses movement to distinguish the user from the background furniture.

To activate the seated mode, you must execute this code:

```
var kinectSensor = KinectSensor.KinectSensors[0];
kinectSensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Seated;
```

Please also consider that in seated mode, the Kinect for Windows SDK will consume more resources as the challenge to detect users is more complicated.

Tracking skeletons

One of the major strengths of the Kinect for Windows SDK is its ability to find the skeleton using a very fast and accurate recognition system that requires no setup, because a learning machine has already been instructed to recognize the skeleton. To be recognized, users simply need to be in front of the sensor, making sure that at least their head and upper body are visible to the Kinect sensor; no specific pose or calibration action needs to be taken for a user to be tracked. When you pass in front of the sensor (at the correct distance, of course), the NUI library will discover your skeleton and will raise an event with useful data about it. In seated mode, you may have to move slightly, as mentioned previously, so that the sensor can distinguish you from the background furniture.

As was also mentioned previously, one or two people can be actively tracked by the sensor at the same time. Kinect will produce passive tracking for up to four additional people in the sensor field of view if there are more than two individuals standing in front of the sensor. When passive tracking is activated, only the skeleton position is computed. Passively tracked skeletons don't have a list of joints.

To activate skeleton tracking in your application, call the following code:

```
var kinectSensor = KinectSensor.KinectSensors[0];
kinectSensor.SkeletonStream.Enable();
```

Note In near mode, you must activate the skeleton tracking system using the following code:

```
var kinectSensor = KinectSensor.KinectSensors[0];
kinectSensor.SkeletonStream.EnableTrackingInNearRange = true;
```

Getting skeleton data

As with depth and color streams, you can retrieve skeleton data using an event or by polling. An application must choose one model or the other; it cannot use both models simultaneously.

```
var skeletondData = kinectSensor.SkeletonStream.OpenNextFrame(500);
```

Following is the code for the event approach:

```
kinectSensor.SkeletonFrameReady += kinectSensor_SkeletonFrameReady;
void kinectSensor _SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
        { }
```

This code will be called every time a depth frame is ready (at 30 FPS).

Browsing skeletons

Using the *SkeletonFrameReadyEventArgs* or the *OpenNextFrame* method, you can get a *SkeletonFrame* object that contains all the available skeletons.

To extract them, you need a method that you will reuse many times. To do this, you can create a new helper class called *Tools* where you can define all of your helpers:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public static class Tools
    {
        public static Skeleton[] GetSkeletons(this SkeletonFrame frame)
        {
            if (frame == null)
                return null;

            var skeletons = new Skeleton[frame.SkeletonArrayLength];
            frame.CopySkeletonDataTo(skeletons);

            return skeletons;
        }
    }
}
```

And then you can use the following code to access the skeletons and browse each of their joints:

```
void kinectSensor_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
{
    SkeletonFrame frame = e.OpenSkeletonFrame();
    if (frame == null)
        return;

    Skeleton[] skeletons = frame.GetSkeletons();

    if (skeletons.All(s => s.TrackingState == SkeletonTrackingState.NotTracked))
        return;
    …
}
```

Each skeleton is defined by a *TrackingState* that indicates if the skeleton is being actively or passively tracked (that is, if it contains joints or is known only by a position).

The *TrackingState* can be one of the following values:

- ***NotTracked*** The skeleton is generated but not found in the field of view of the sensor.
- ***PositionOnly*** The skeleton is passively tracked.
- ***Tracked*** The skeleton is actively tracked.

Each tracked skeleton will have a collection *Joints*, which is the set of control points.

Furthermore, each skeleton contains a *TrackingID* property. The tracking ID is guaranteed to remain consistently applied to the same person in front of the scanner for as long as he or she remains

in the field of view. A specific tracking ID is guaranteed to remain at the same index in the skeleton data array for as long as the tracking ID is in use. Note that if a user leaves the scene and comes back, that user will receive a new tracking ID chosen randomly—it will not be related to the tracking ID that the same user had when he or she left the scene.

Applications can also use the tracking ID to maintain the coherency of the identification of the people who are seen by the scanner.

By default, skeletal tracking will select the first two recognized users in the field of view. If you prefer, you can program the application to override the default behavior by defining custom logic for selecting which users to track, such as choosing the user closest to the camera or a user who is raising his or her hand.

To do so, applications can cycle through the proposed skeletons, selecting those that fit the criteria of the custom logic and then pass their tracking IDs to the skeletal tracking APIs for full tracking:

```
kinectSensor.SkeletonStream.AppChoosesSkeletons = true;
kinectSensor.SkeletonStream.ChooseSkeletons(1, 5);
```

When the application has control over which users to track, the skeletal tracking system will not take it back—if the user goes out of the screen, it is up to the application to select a new user to track.

Applications can also select to track only one skeleton or no skeletons at all by passing a null tracking ID to the skeletal tracking APIs.

Finally, each tracked skeleton, whether passive or active, has a position value that is the center of mass of the associated person. The position is composed of a 3D coordinate (x, y, z) and a confidence level (W).

You'll see in the coming chapters that the skeleton is the main tool for handling gestures and postures with Kinect.

PART II

Integrate Kinect in your application

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 *INotifyPropertyChanged* 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 *WritePixels* 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 *ColorStreamManager* 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.

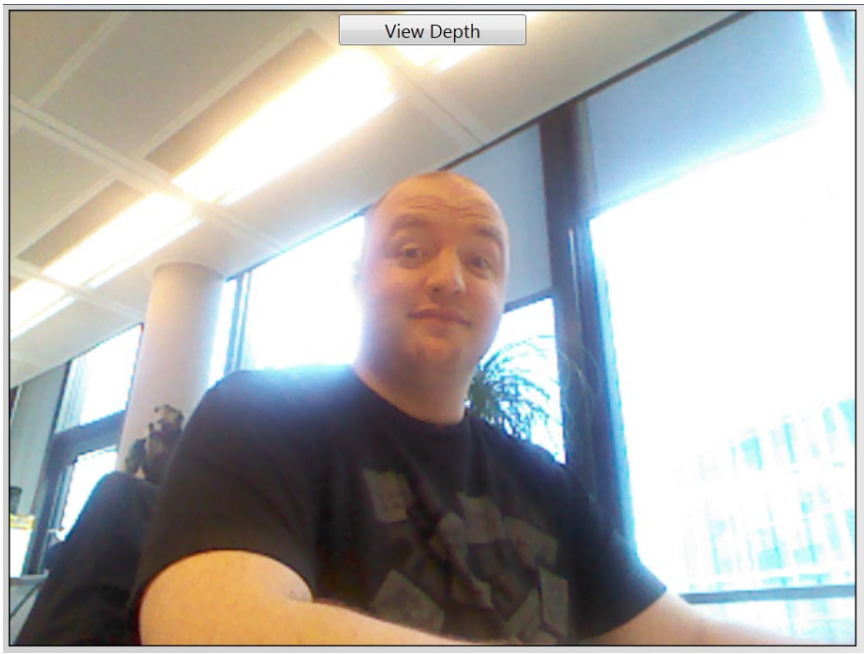

FIGURE 3-1 Displaying the Kinect color stream with WPF.

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.)

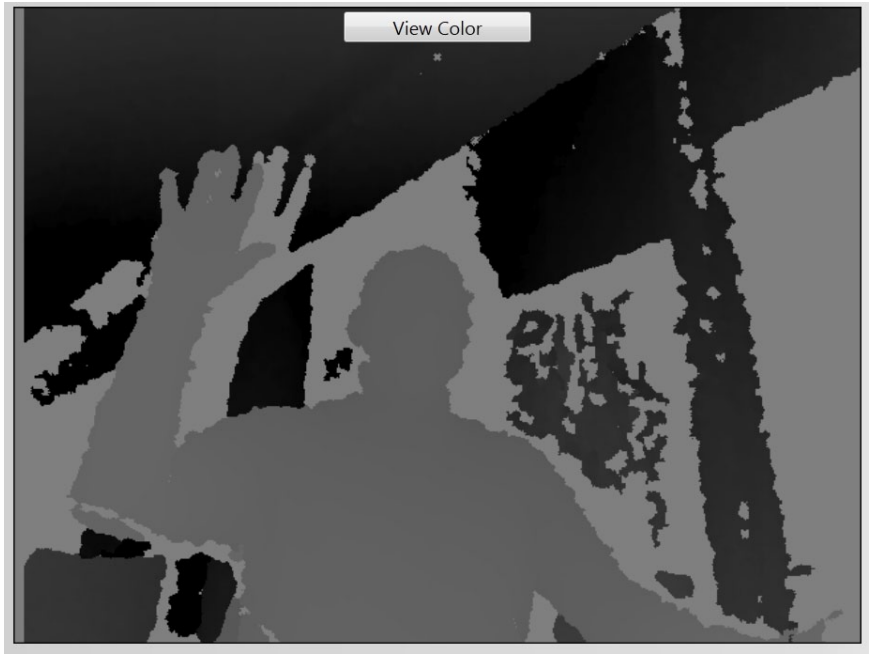

FIGURE 3-2 The depth stream display without a user detected.

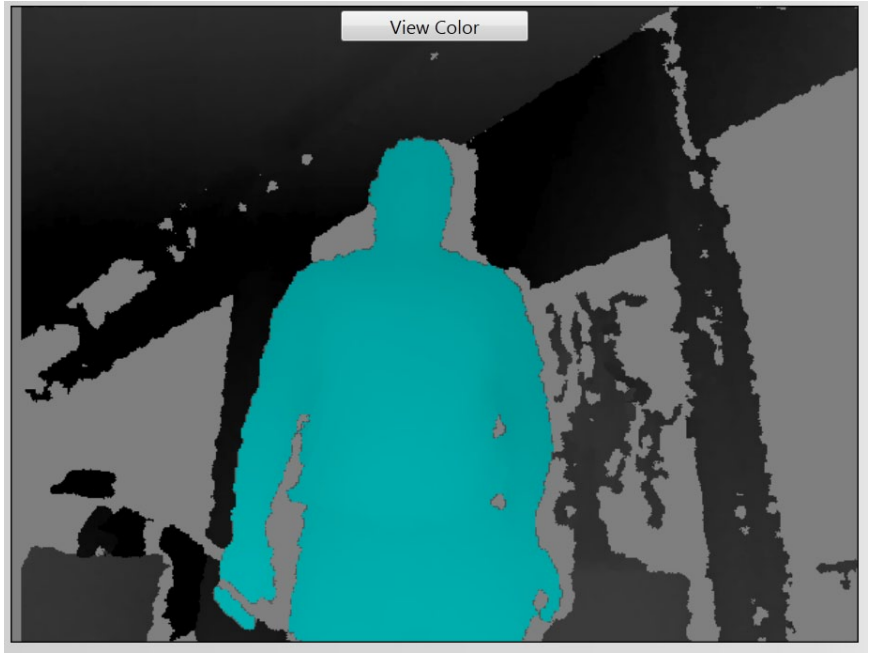

FIGURE 3-3 The depth stream display with a user detected. (A specific color is used where the user is detected, but this appears as light gray to readers of the print book.)

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.

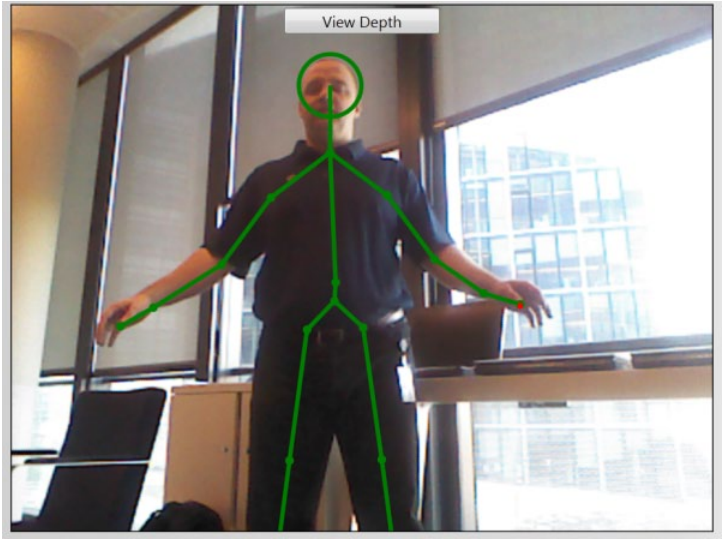

FIGURE 3-5 Displaying the skeleton data.

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 4

Recording and playing a Kinect session

When you are developing applications with Kinect for Windows, you will probably spend a lot of time getting up to test the code you have written. Furthermore, you will need to replay some test cases to validate your algorithms. So a tool that can enable the replay of a complete Kinect session (with color, depth, and skeleton data) is a necessity.

In this chapter, you will write such a tool. You will also add a voice controller that will allow you to be totally autonomous during debugging sessions, because it lets you command both the recorder and the player with your voice alone.

Kinect Studio

But before going back to code, let's talk about Kinect Studio.

Starting with SDK 1.5, Microsoft launches a companion to the Kinect for Windows SDK: the Kinect for Windows Toolkit. Among other new features, the toolkit has a new application called Kinect Studio. Kinect Studio is a tool that allows you to record and play back Kinect data to aid in development. You can record clips of users and then replay those clips at a later time for development and test purposes.

Kinect Studio must be used in conjunction with your own application. That is, you have to start your application, then start Kinect Studio, and finally launch the playback of a clip to use Kinect Studio's features.

More Info You can find documentation related to Kinect Studio at *http://msdn.microsoft.com/en-us/library/hh855395*.

So you may be asking yourself why you should have to write a code to do the same thing, as you will do in this chapter. Even with this new tool, however, you will find the code introduced in this chapter useful for several reasons.

First, creating this code is part of a learning process that will help you improve your understanding of Kinect for Windows SDK. Also, you will find that there are times when you will not be able to install

the Kinect for Windows SDK (on end user machines, for instance), but you will still need to record or replay a clip for debugging purposes. Finally, you may want to use the code in this chapter for documentation purposes by integrating a replay system for prerecorded sessions in your application to help users understand how to use your application.

Recording Kinect data

To record Kinect data, you create a class called *KinectRecorder* that handles all three kinds of data (color, depth, and skeletons). This class can record the Kinect data structures and then replay them accurately.

The *KinectRecorder* class uses a single file output that contains all required data (the data from each source is interlaced to form a single file).

The KinectRecorder aggregates three classes (one for each kind of data):

- *ColorRecorder*
- *DepthRecorder*
- *SkeletonRecorder*

Each subrecorder appends its own data to the main stream and includes a small header for identification, as shown in Figure 4-1.

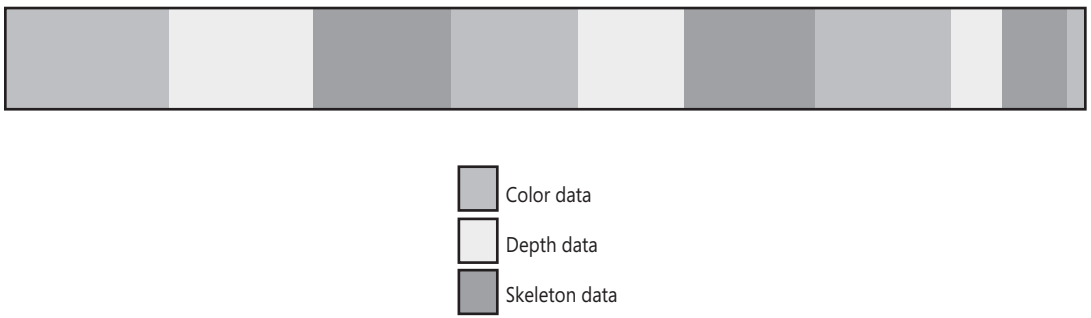

FIGURE 4-1 The structure of a Kinect session record file.

The header is only an integer representing one of the following values:

```
[FlagsAttribute]
public enum KinectRecordOptions
{
    Color = 1,
    Depth = 2,
    Skeletons = 4
}
```

You have to include this enum in your project so that you can use the code created during this chapter.

Later in this chapter you will see how the data is decoded by a specific class (*KinectReplay*).

Important Note the need for a reference time. Each recorder must keep a reference time variable and add it to the saved stream to allow the player to reproduce the data at the same frequency.

Recording the color stream

The first stream to record is the color stream, and to do so, you simply indicate a stream of bytes to write, along with some format information:

```
using System;
using System.IO;
using Microsoft.Kinect;

class ColorRecorder
{
    DateTime referenceTime;
    readonly BinaryWriter writer;

    internal ColorRecorder(BinaryWriter writer)
    {
        this.writer = writer;
        referenceTime = DateTime.Now;
    }

    public void Record(ColorImageFrame frame)
    {
        // Header
        writer.Write((int)KinectRecordOptions.Color);

        // Data
        TimeSpan timeSpan = DateTime.Now.Subtract(referenceTime);
        referenceTime = DateTime.Now;
        writer.Write((long)timeSpan.TotalMilliseconds);
        writer.Write(frame.BytesPerPixel);
        writer.Write((int)frame.Format);
        writer.Write(frame.Width);
        writer.Write(frame.Height);

        writer.Write(frame.FrameNumber);

        // Bytes
        writer.Write(frame.PixelDataLength);
        byte[] bytes = new byte[frame.PixelDataLength];
        frame.CopyPixelDataTo(bytes);
        writer.Write(bytes);
    }
}
```

You save specific information (such as width, height, format, etc.) and use the *CopyPixelDataTo* method of the frame to get a byte array you can write to the stream.

Recording the depth stream

As with the color stream, recording the depth stream simply involves recording a bytes array:

```
using System;
using System.IO;
using Microsoft.Kinect;

class DepthRecorder
{
    DateTime referenceTime;
    readonly BinaryWriter writer;

    internal DepthRecorder(BinaryWriter writer)
    {
        this.writer = writer;
        referenceTime = DateTime.Now;
    }

    public void Record(DepthImageFrame frame)
    {
        // Header
        writer.Write((int)KinectRecordOptions.Depth);

        // Data
        TimeSpan timeSpan = DateTime.Now.Subtract(referenceTime);
        referenceTime = DateTime.Now;
        writer.Write((long)timeSpan.TotalMilliseconds);
        writer.Write(frame.BytesPerPixel);
        writer.Write((int)frame.Format);
        writer.Write(frame.Width);
        writer.Write(frame.Height);

        writer.Write(frame.FrameNumber);

        // Bytes
        short[] shorts = new short[frame.PixelDataLength];
        frame.CopyPixelDataTo(shorts);
        writer.Write(shorts.Length);
        foreach (short s in shorts)
        {
            writer.Write(s);
        }
    }
}
```

The main difference between recording the color stream and recording the depth stream is that the *CopyPixelDataTo* of the depth frame uses a short array, so you must cycle through all indices of the array to save them individually.

Recording the skeleton frames

Recording the skeleton frames is somewhat more complex because structured data must be serialized. Indeed, at a specific rate (30 times per second if everything is okay), the Kinect sensor produces a *SkeletonFrame* that is composed of

- A frame number
- A floor clip plane (composed of a tuple of 4 floats) that defines the x, y, z, and w coordinates of a vector that is used internally to clip below the floor
- An array of *Skeleton* objects

By chance, the *Skeleton* class is marked as *[Serializable]*, so you only need to write some additional information before calling a serialization of the skeletons array.

Note The following code continues to use the *Tools* class developed in previous chapters.

```
using System;
using System.IO;
using Microsoft.Kinect;
using System.Runtime.Serialization.Formatters.Binary;

class SkeletonRecorder
{
    DateTime referenceTime;
    readonly BinaryWriter writer;

    internal SkeletonRecorder(BinaryWriter writer)
    {
        this.writer = writer;
        referenceTime = DateTime.Now;
    }

    public void Record(SkeletonFrame frame)
    {
        // Header
        writer.Write((int)KinectRecordOptions.Skeletons);

        // Data
        TimeSpan timeSpan = DateTime.Now.Subtract(referenceTime);
        referenceTime = DateTime.Now;
        writer.Write((long)timeSpan.TotalMilliseconds);
        writer.Write(frame.FloorClipPlane.Item1);
        writer.Write(frame.FloorClipPlane.Item2);
        writer.Write(frame.FloorClipPlane.Item3);
        writer.Write(frame.FloorClipPlane.Item4);

        writer.Write(frame.FrameNumber);

        // Skeletons
        Skeleton[] skeletons = frame.GetSkeletons();


        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(writer.BaseStream, skeletons);
    }
}
```

You save each frame using the following process:

- Write the header (*KinectRecordOptions.Skeletons*).
- Compute the delta from the current time to the reference time and write it.
- Write the floor clip plane.
- Write the frame number.
- Serialize the skeletons array.

Putting it all together

The *KinectRecorder* class aggregates all the recording classes to provide a unique entry point for developers. It contains a reference for each of the recording classes as well as a stream and a *BinaryWriter*:

```
Stream recordStream;
readonly BinaryWriter writer;

// Recorders
readonly ColorRecorder colorRecoder;
readonly DepthRecorder depthRecorder;
readonly SkeletonRecorder skeletonRecorder;

public KinectRecordOptions Options { get; set; }
```

According to the specified options, the constructor instantiates the subclasses and creates the stream:

```
// Constructor
public KinectRecorder(KinectRecordOptions options, Stream stream)
{
    Options = options;

    recordStream = stream;
    writer = new BinaryWriter(recordStream);

    writer.Write((int)Options);

    if ((Options & KinectRecordOptions.Color) != 0)
    {
        colorRecoder = new ColorRecorder(writer);
    }
    if ((Options & KinectRecordOptions.Depth) != 0)
    {
        depthRecorder = new DepthRecorder(writer);
    }
    if ((Options & KinectRecordOptions.Skeletons) != 0)
    {
        skeletonRecorder = new SkeletonRecorder(writer);
    }
}
```

Finally, it provides three methods to record specific frames (color, depth, and skeleton):

```
public void Record(SkeletonFrame frame)
{
    if (writer == null)
        throw new Exception("This recorder is stopped");

    if (skeletonRecorder == null)
        throw new Exception("Skeleton recording is not activated on this KinectRecorder");

    skeletonRecorder.Record(frame);
}

public void Record(ColorImageFrame frame)
{
    if (writer == null)
        throw new Exception("This recorder is stopped");

    if (colorRecoder == null)
        throw new Exception("Color recording is not activated on this KinectRecorder");

    colorRecoder.Record(frame);
}

public void Record(DepthImageFrame frame)
{
    if (writer == null)
        throw new Exception("This recorder is stopped");

    if (depthRecorder == null)
        throw new Exception("Depth recording is not activated on this KinectRecorder");

    depthRecorder.Record(frame);
}
```

The final class code is the following:

```
using System;
using System.IO;
using Microsoft.Kinect;


public class KinectRecorder
{
    Stream recordStream;
    readonly BinaryWriter writer;

    // Recorders
    readonly ColorRecorder colorRecoder;
    readonly DepthRecorder depthRecorder;
    readonly SkeletonRecorder skeletonRecorder;

    public KinectRecordOptions Options { get; set; }

    // Ctr
    public KinectRecorder(KinectRecordOptions options, Stream stream)
```

```
{
    Options = options;

    recordStream = stream;
    writer = new BinaryWriter(recordStream);

    writer.Write((int)Options);

    if ((Options & KinectRecordOptions.Color) != 0)
    {
        colorRecoder = new ColorRecorder(writer);
    }
    if ((Options & KinectRecordOptions.Depth) != 0)
    {
        depthRecorder = new DepthRecorder(writer);
    }
    if ((Options & KinectRecordOptions.Skeletons) != 0)
    {
        skeletonRecorder = new SkeletonRecorder(writer);
    }
}

public void Record(SkeletonFrame frame)
{
    if (writer == null)
        throw new Exception("This recorder is stopped");

    if (skeletonRecorder == null)
        throw new Exception("Skeleton recording is not activated on this KinectRecorder");

    skeletonRecorder.Record(frame);
}

public void Record(ColorImageFrame frame)
{
    if (writer == null)
        throw new Exception("This recorder is stopped");

    if (colorRecoder == null)
        throw new Exception("Color recording is not activated on this KinectRecorder");

    colorRecoder.Record(frame);
}

public void Record(DepthImageFrame frame)
{
    if (writer == null)
        throw new Exception("This recorder is stopped");

    if (depthRecorder == null)
        throw new Exception("Depth recording is not activated on this KinectRecorder");

    depthRecorder.Record(frame);
}

public void Stop()
{
```

```
            if (writer == null)
                throw new Exception("This recorder is already stopped");

            writer.Close();
            writer.Dispose();

            recordStream.Dispose();
            recordStream = null;
        }
    }
```

Replaying Kinect data

To replay previously saved data, you must write a class named *KinectReplay*. It is responsible for centralizing the replay system to provide recorded events to your application. Replaying color, depth, or the skeleton frames is essentially the same thing.

A frame must be a least defined by a *FrameNumber* and a *TimeStamp*. It also must provide a *CreateFromReader* method to allow loading from a stream:

```
using System.IO;

public abstract class ReplayFrame
{
    public int FrameNumber { get; protected set; }
    public long TimeStamp { get; protected set; }

    internal abstract void CreateFromReader(BinaryReader reader);
}
```

The replay system is based on a task (from the Task Parallel Library [TPL]) to handle the time between subsequent frames. This task causes the system to sleep for a given time, after which the task raises an event for the *KinectReader* to signal that a new frame is available. It provides an *AddFrame* internal method to allow the *KinectReplay* class to fill it with saved frames.

A *Stop* method cancels the token used inside the task to exit properly:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

class ReplaySystem<T>  where T:ReplayFrame, new()
{
    internal event Action<T> FrameReady;
    readonly List<T> frames = new List<T>();

    CancellationTokenSource cancellationTokenSource;
    public bool IsFinished
    {
        get;
```

```
        private set;
    }

    internal void AddFrame(BinaryReader reader)
    {
        T frame = new T();

        frame.CreateFromReader(reader);

        frames.Add(frame);
    }

    public void Start()
    {
        Stop();
        IsFinished = false;

        cancellationTokenSource = new CancellationTokenSource();

        CancellationToken token = cancellationTokenSource.Token;

        Task.Factory.StartNew(() =>
        {
            foreach (T frame in frames)
            {
                Thread.Sleep(TimeSpan.FromMilliseconds(frame.TimeStamp));

                if (token.IsCancellationRequested)
                    break;

                if (FrameReady != null)
                    FrameReady(frame);
            }
            IsFinished = true;
        }, token);
    }

    public void Stop()
    {
        if (cancellationTokenSource == null)
            return;

        cancellationTokenSource.Cancel();
    }
}
```

The previous code defines a generic class called *ReplaySystem<T>*. This class uses a generic parameter, which must inherit from *ReplayFrame*. Then the main job for each stream is to provide a correct frame for the *ReplaySystem<T>*.

Replaying color streams

The important thing to remember is that the *ColorImageFrame* class has a private constructor, which prevents you from creating a new instance. The same constraint applies to *DepthImageFrame* and *SkeletonFrame*.

The replay system must then have an impostor *ReplayColorImageFrame* class that can be cast from a *ColorImageFrame*. All of your code should now refer to *ReplayColorImageFrame* instead of the original, in order to work well with live and replayed data.

Obviously, *ReplayColorImageFrame* must also inherit from *ReplayFrame*:

```
using System.IO;
using Microsoft.Kinect;

namespace Kinect.Toolbox.Record
{
    public class ReplayColorImageFrame : ReplayFrame
    {
        readonly ColorImageFrame internalFrame;
        long streamPosition;
        Stream stream;

        public int Width { get; private set; }
        public int Height { get; private set; }
        public int BytesPerPixel { get; private set; }
        public ColorImageFormat Format { get; private set; }
        public int PixelDataLength { get; set; }

        public ReplayColorImageFrame(ColorImageFrame frame)
        {
            Format = frame.Format;
            BytesPerPixel = frame.BytesPerPixel;
            FrameNumber = frame.FrameNumber;
            TimeStamp = frame.Timestamp;
            Width = frame.Width;
            Height = frame.Height;

            PixelDataLength = frame.PixelDataLength;
            internalFrame = frame;
        }

        public ReplayColorImageFrame()
        {

        }

        internal override void CreateFromReader(BinaryReader reader)
        {
            TimeStamp = reader.ReadInt64();
            BytesPerPixel = reader.ReadInt32();
            Format = (ColorImageFormat)reader.ReadInt32();
            Width = reader.ReadInt32();
            Height = reader.ReadInt32();
            FrameNumber = reader.ReadInt32();
```

```
            PixelDataLength = reader.ReadInt32();

            stream = reader.BaseStream;
            streamPosition = stream.Position;

            stream.Position += PixelDataLength;
        }

        public void CopyPixelDataTo(byte[] pixelData)
        {
            if (internalFrame != null)
            {
                internalFrame.CopyPixelDataTo(pixelData);
                return;
            }

            long savedPosition = stream.Position;
            stream.Position = streamPosition;

            stream.Read(pixelData, 0, PixelDataLength);

            stream.Position = savedPosition;
        }

        public static implicit operator ReplayColorImageFrame(ColorImageFrame frame)
        {
            return new ReplayColorImageFrame(frame);
        }
    }
}
```

As you can see, the *ReplayColorImageFrame* class provides an implicit cast operator from *ColorImageFrame*.

To use this class, you have to write code that simply refers to it:

```
void ProcessFrame(ColorImageFrame frame)
```

For instance, this method can be called with a *ReplayColorImageFrame* or a standard *ColorImageFrame* (thanks to the implicit cast operator).

Another important element to remember here is the *CopyPixelDataTo* method. If the *ReplayColorImageFrame* was built using a *ColorImageFrame*, you simply use the standard *CopyToPixelDataTo* method. But if the frame came from a stream, you have to save the given stream and the position of the bytes array inside (it would be handy to read and save a copy of the buffer in memory, but for obvious performance and memory consumption reasons, it is better to keep a link to the stream).

Replaying depth streams

The *ReplayFrame* for depth stream looks like the *ReplayColorImageFrame*:

```
using System.IO;
using Microsoft.Kinect;

public class ReplayDepthImageFrame : ReplayFrame
{
    readonly DepthImageFrame internalFrame;
    long streamPosition;
    Stream stream;
    BinaryReader streamReader;

    public int Width { get; private set; }
    public int Height { get; private set; }
    public int BytesPerPixel { get; private set; }
    public DepthImageFormat Format { get; private set; }
    public int PixelDataLength { get; set; }

    public ReplayDepthImageFrame(DepthImageFrame frame)
    {
        Format = frame.Format;
        BytesPerPixel = frame.BytesPerPixel;
        FrameNumber = frame.FrameNumber;
        TimeStamp = frame.Timestamp;
        Width = frame.Width;
        Height = frame.Height;

        PixelDataLength = frame.PixelDataLength;
        internalFrame = frame;
    }

    public ReplayDepthImageFrame()
    {

    }

    internal override void CreateFromReader(BinaryReader reader)
    {
        TimeStamp = reader.ReadInt64();
        BytesPerPixel = reader.ReadInt32();
        Format = (DepthImageFormat)reader.ReadInt32();
        Width = reader.ReadInt32();
        Height = reader.ReadInt32();
        FrameNumber = reader.ReadInt32();

        PixelDataLength = reader.ReadInt32();

        stream = reader.BaseStream;
        streamReader = reader;
        streamPosition = stream.Position;

        stream.Position += PixelDataLength * 2;
    }
```

```
    public void CopyPixelDataTo(short[] pixelData)
    {
        if (internalFrame != null)
        {
            internalFrame.CopyPixelDataTo(pixelData);
            return;
        }

        long savedPosition = stream.Position;
        stream.Position = streamPosition;

        for (int index = 0; index < PixelDataLength; index++)
        {
            pixelData[index] = streamReader.ReadInt16();
        }

        stream.Position = savedPosition;
    }

    public static implicit operator ReplayDepthImageFrame(DepthImageFrame frame)
    {
        return new ReplayDepthImageFrame(frame);
    }
}
```

You have to deal with the array of shorts produced by depth streams, and you have to keep a reference to the *BinaryReader* to be able to read all shorts in the *CopyPixelDataTo* method.

Except on this point, the behavior is the same with an implicit cast operator from *DepthImageFrame*.

Replaying skeleton frames

The *ReplaySkeletonFrame* is even simpler. You don't have to handle the *CopyPixelDataTo* so the code is shorter:

```
using System;
using System.IO;
using Microsoft.Kinect;
using System.Runtime.Serialization.Formatters.Binary;

public class ReplaySkeletonFrame : ReplayFrame
{
    public Tuple<float, float, float, float> FloorClipPlane { get; private set; }
    public Skeleton[] Skeletons { get; private set; }

    public ReplaySkeletonFrame(SkeletonFrame frame)
    {
        FloorClipPlane = frame.FloorClipPlane;
        FrameNumber = frame.FrameNumber;
        TimeStamp = frame.Timestamp;
        Skeletons = frame.GetSkeletons();
    }
```

```
    public ReplaySkeletonFrame()
    {

    }

    internal override void CreateFromReader(BinaryReader reader)
    {
        TimeStamp = reader.ReadInt64();
        FloorClipPlane = new Tuple<float, float, float, float>(
            reader.ReadSingle(), reader.ReadSingle(),
            reader.ReadSingle(), reader.ReadSingle());

        FrameNumber = reader.ReadInt32();

        BinaryFormatter formatter = new BinaryFormatter();
        Skeletons = (Skeleton[]) formatter.Deserialize(reader.BaseStream);
    }

    public static implicit operator ReplaySkeletonFrame(SkeletonFrame frame)
    {
        return new ReplaySkeletonFrame(frame);
    }
}
```

Putting it all together

The *KinectReplay* class is responsible for aggregating all these classes inside a unique hub. It contains three events for users:

```
// Events
public event EventHandler<ReplayColorImageFrameReadyEventArgs> ColorImageFrameReady;
public event EventHandler<ReplayDepthImageFrameReadyEventArgs> DepthImageFrameReady;
public event EventHandler<ReplaySkeletonFrameReadyEventArgs> SkeletonFrameReady;
```

The events use some specifics event arguments defined as follows:

```
using System;

namespace Kinect.Toolbox.Record
{
    public class ReplayColorImageFrameReadyEventArgs : EventArgs
    {
        public ReplayColorImageFrame ColorImageFrame { get; set; }
    }

    public class ReplayDepthImageFrameReadyEventArgs : EventArgs
    {
        public ReplayDepthImageFrame DepthImageFrame { get; set; }
    }

    public class ReplaySkeletonFrameReadyEventArgs : EventArgs
    {
        public ReplaySkeletonFrame SkeletonFrame { get; set; }
    }
}
```

And of course it contains an instance of *ReplaySystem<T>* for each kind of stream:

```
// Replay
ReplaySystem<ReplayColorImageFrame> colorReplay;
ReplaySystem<ReplayDepthImageFrame> depthReplay;
ReplaySystem<ReplaySkeletonFrame> skeletonReplay;
```

It also requires a stream, a *BinaryReader*, and a synchronization context to raise events in the correct context:

```
BinaryReader reader;
Stream stream;
readonly SynchronizationContext synchronizationContext;
```

A *Started* auto-property is defined to export class status:

```
public bool Started { get; internal set; }
```

The class constructor uses a stream to initialize. First it determines the data available in the stream and creates the associated *ReplaySystem<T>* accordingly.

Finally, it browses the entire stream looking for data headers and adds frames to the right *ReplaySystem<T>*:

```
public KinectReplay(Stream stream)
{
    this.stream = stream;
    reader = new BinaryReader(stream);

    synchronizationContext = SynchronizationContext.Current;

    KinectRecordOptions options = (KinectRecordOptions) reader.ReadInt32();

    if ((options & KinectRecordOptions.Color) != 0)
    {
        colorReplay = new ReplaySystem<ReplayColorImageFrame>();
    }
    if ((options & KinectRecordOptions.Depth) != 0)
    {
        depthReplay = new ReplaySystem<ReplayDepthImageFrame>();
    }
    if ((options & KinectRecordOptions.Skeletons) != 0)
    {
        skeletonReplay = new ReplaySystem<ReplaySkeletonFrame>();
    }

    while (reader.BaseStream.Position != reader.BaseStream.Length)
    {
        KinectRecordOptions header = (KinectRecordOptions)reader.ReadInt32();
        switch (header)
        {
            case KinectRecordOptions.Color:
                colorReplay.AddFrame(reader);
                break;
            case KinectRecordOptions.Depth:
```

```
                depthReplay.AddFrame(reader);
                break;
            case KinectRecordOptions.Skeletons:
                skeletonReplay.AddFrame(reader);
                break;
        }
    }
}
```

The *Start* method launches all internal components and provides a link between internal events (*FrameReady*) and correct external events:

```
public void Start()
{
    if (Started)
        throw new Exception("KinectReplay already started");

    Started = true;

    if (colorReplay != null)
    {
        colorReplay.Start();
        colorReplay.FrameReady += frame => synchronizationContext.Send(state =>
        {
            if (ColorImageFrameReady != null)
                ColorImageFrameReady(this,
new ReplayColorImageFrameReadyEventArgs { ColorImageFrame = frame });
        }, null);
    }

    if (depthReplay != null)
    {
        depthReplay.Start();
        depthReplay.FrameReady += frame => synchronizationContext.Send(state =>
        {
            if (DepthImageFrameReady != null)
                DepthImageFrameReady(this,
new ReplayDepthImageFrameReadyEventArgs { DepthImageFrame = frame });
        }, null);
    }

    if (skeletonReplay != null)
    {
        skeletonReplay.Start();
        skeletonReplay.FrameReady += frame => synchronizationContext.Send(state =>
        {
            if (SkeletonFrameReady != null)
                SkeletonFrameReady(this,
new ReplaySkeletonFrameReadyEventArgs { SkeletonFrame = frame });
        }, null);
    }
}
```

The final code is:

```
using System;
```

```
using System.IO;
using System.Threading;

public class KinectReplay : IDisposable
{
    BinaryReader reader;
    Stream stream;
    readonly SynchronizationContext synchronizationContext;

    // Events
    public event EventHandler<ReplayColorImageFrameReadyEventArgs> ColorImageFrameReady;
    public event EventHandler<ReplayDepthImageFrameReadyEventArgs> DepthImageFrameReady;
    public event EventHandler<ReplaySkeletonFrameReadyEventArgs> SkeletonFrameReady;


    // Replay
    ReplaySystem<ReplayColorImageFrame> colorReplay;
    ReplaySystem<ReplayDepthImageFrame> depthReplay;
    ReplaySystem<ReplaySkeletonFrame> skeletonReplay;

    public bool Started { get; internal set; }

    public bool IsFinished
    {
        get
        {
            if (colorReplay != null && !colorReplay.IsFinished)
                return false;

            if (depthReplay != null && !depthReplay.IsFinished)
                return false;

            if (skeletonReplay != null && !skeletonReplay.IsFinished)
                return false;

            return true;
        }
    }

    public KinectReplay(Stream stream)
    {
        this.stream = stream;
        reader = new BinaryReader(stream);

        synchronizationContext = SynchronizationContext.Current;

        KinectRecordOptions options = (KinectRecordOptions)reader.ReadInt32();

        if ((options & KinectRecordOptions.Color) != 0)
        {
            colorReplay = new ReplaySystem<ReplayColorImageFrame>();
        }
        if ((options & KinectRecordOptions.Depth) != 0)
        {
            depthReplay = new ReplaySystem<ReplayDepthImageFrame>();
        }
        if ((options & KinectRecordOptions.Skeletons) != 0)
```

```
        {
            skeletonReplay = new ReplaySystem<ReplaySkeletonFrame>();
        }

        while (reader.BaseStream.Position != reader.BaseStream.Length)
        {
            KinectRecordOptions header = (KinectRecordOptions)reader.ReadInt32();
            switch (header)
            {
                case KinectRecordOptions.Color:
                    colorReplay.AddFrame(reader);
                    break;
                case KinectRecordOptions.Depth:
                    depthReplay.AddFrame(reader);
                    break;
                case KinectRecordOptions.Skeletons:
                    skeletonReplay.AddFrame(reader);
                    break;
            }
        }
    }

    public void Start()
    {
        if (Started)
            throw new Exception("KinectReplay already started");

        Started = true;

        if (colorReplay != null)
        {
            colorReplay.Start();
            colorReplay.FrameReady += frame => synchronizationContext.Send(state =>
            {
                if (ColorImageFrameReady != null)
                    ColorImageFrameReady(this,
new ReplayColorImageFrameReadyEventArgs { ColorImageFrame = frame });
            }, null);
        }

        if (depthReplay != null)
        {
            depthReplay.Start();
            depthReplay.FrameReady += frame => synchronizationContext.Send(state =>
            {
                if (DepthImageFrameReady != null)
                    DepthImageFrameReady(this,
new ReplayDepthImageFrameReadyEventArgs { DepthImageFrame = frame });
            }, null);
        }

        if (skeletonReplay != null)
        {
            skeletonReplay.Start();
            skeletonReplay.FrameReady += frame => synchronizationContext.Send(state =>
            {
                if (SkeletonFrameReady != null)
```

```
                    SkeletonFrameReady(this,
new ReplaySkeletonFrameReadyEventArgs { SkeletonFrame = frame });
            }, null);
        }
    }

    public void Stop()
    {
        if (colorReplay != null)
        {
            colorReplay.Stop();
        }

        if (depthReplay != null)
        {
            depthReplay.Stop();
        }

        if (skeletonReplay != null)
        {
            skeletonReplay.Stop();
        }

        Started = false;
    }

    public void Dispose()
    {
        Stop();

        colorReplay = null;
        depthReplay = null;
        skeletonReplay = null;

        if (reader != null)
        {
            reader.Dispose();
            reader = null;
        }

        if (stream != null)
        {
            stream.Dispose();
            stream = null;
        }
    }
}
```

With this class, you're now ready to replay a previously recorded Kinect session and integrate it with your existing code. For instance, if you have a *ProcessFrame* method that takes a *ColorImageFrame* in parameter, you just have to change the type of the parameter to *ReplayColorImageFrame,* and without any more changes, your method accepts real-time data (with *ColorImageFrame,* which will be implicitly cast to *ReplayColorImageFrame*) and offline data (with *ReplayColorImageFrame*):

```
private void LaunchReplay()
{
    OpenFileDialog openFileDialog =
new OpenFileDialog { Title = "Select filename", Filter = "Replay files|*.replay" };

    if (openFileDialog.ShowDialog() == true)
    {
        Stream recordStream = File.OpenRead(openFileDialog.FileName);

        replay = new KinectReplay(recordStream);

        replay.ColorImageFrameReady += replay_ColorImageFrameReady;

        replay.Start();
    }
}
```

This method refers to a *replay_ColorImageFrameReady* event:

```
void replay_ColorImageFrameReady (object sender, ReplayColorImageFrameReadyEventArgse)
{
    ProcessFrame(e. ColorImageFrame);
}
```

The process frame must be changed from:

```
void ProcessFrame(ColorImageFrame frame)
{
   …
}
```

to:

```
void ProcessFrame(ReplayColorImageFrame frame)
{
   …
}
```

The process works the same way with the depth and skeleton frames.

Controlling the record system with your voice

You now have a nifty record/replay system for developing and debugging Kinect for Windows applications. But when you want to create new records, you need to have someone else ready to click to launch and stop the record (because *you* are standing up in front of your sensor).

This is where the microphone array of Kinect is particularly useful. With the help of Microsoft Speech API, you will write a voice command system that will start and end the record system for you.

The *VoiceCommander* class is built around the *AudioSource* of the Kinect sensor. This source is fully compatible with *Microsoft.Speech* (installed with the Kinect for Windows SDK). Indeed, it can provide a stream used by the *SpeechRecognitionEngine* class to detect words.

To do this, you must first get the audio source object and find a voice recognizer between all installed recognizers in the system (search for a recognizer containing "Kinect" in its name):

```
KinectAudioSource source = kinectSensor.AudioSource;

Func<RecognizerInfo, bool> matchingFunc = r =>
{
    string value;
    r.AdditionalInfo.TryGetValue("Kinect", out value);
    return
        "True".Equals(value, StringComparison.InvariantCultureIgnoreCase) &&
        "en-US".Equals(r.Culture.Name, StringComparison.InvariantCultureIgnoreCase);
};

var recognizerInfo = SpeechRecognitionEngine.InstalledRecognizers().Where(matchingFunc).
FirstOrDefault();

if (recognizerInfo == null)
    return;
```

When you find the voice recognizer, you can create the *SpeechRecognitionEngine* object and start building your grammar (the list of recognized words and sentences).

```
speechRecognitionEngine = new SpeechRecognitionEngine(recognizerInfo.Id);

var gb = new GrammarBuilder { Culture = recognizerInfo.Culture };
var choices = new[]{"record", "stop"};
gb.Append(choices);

var grammar = new Grammar(gb);

speechRecognitionEngine.LoadGrammar(grammar);
```

You can then configure the audio stream as follows:

```
source.AutomaticGainControlEnabled = false;
source.BeamAngleMode = BeamAngleMode.Adaptive;
```

With these lines of code, you deactivate the automatic gain control and set the Kinect microphone array to an auto adaptive mode. You can change and tweak these values to adapt to your own environment.

With the *SpeechRecognitionEngine* object, the configured stream, and your grammar, you simply have to start the stream and wait for the system to detect a word with a good confidence level:

```
using (Stream sourceStream = source.Start())
{
    speechRecognitionEngine.SetInputToAudioStream(sourceStream,
        new SpeechAudioFormatInfo(EncodingFormat.Pcm, 16000, 16, 1, 32000, 2, null));

    while (isRunning)
    {
        RecognitionResult result = speechRecognitionEngine.Recognize();
        if (result != null && OrderDetected != null && result.Confidence > 0.7)
```

```
            OrderDetected(result.Text);
    }
}
```

The confidence level in the preceding code is set arbitrarily to 0.7 (70%), but you can define your own level:

```
if (result != null && OrderDetected != null && result.Confidence > 0.7)
```

The final code is as follows:

```
using System;
using System.IO;
using System.Linq;
using System.Threading;
using Microsoft.Kinect;
using Microsoft.Speech.AudioFormat;
using Microsoft.Speech.Recognition;
public class VoiceCommander
{
    Thread workingThread;
    readonly Choices choices;
    bool isRunning;
    SpeechRecognitionEngine speechRecognitionEngine;
    private KinectSensor kinectSensor;

    public event Action<string> OrderDetected;

    public VoiceCommander(params string[] orders)
    {
        choices = new Choices();
        choices.Add(orders);
    }

    public void Start(KinectSensor sensor)
    {
        if (isRunning)
            throw new Exception("VoiceCommander is already running");

        isRunning = true;
        kinectSensor = sensor;
        workingThread = new Thread(Record) {IsBackground = true};
        workingThread.Start();
    }

    void Record()
    {
        KinectAudioSource source = kinectSensor.AudioSource;

        Func<RecognizerInfo, bool> matchingFunc = r =>
        {
            string value;
            r.AdditionalInfo.TryGetValue("Kinect", out value);
            return
                "True".Equals(value, StringComparison.InvariantCultureIgnoreCase) &&
                "en-US".Equals(r.Culture.Name, StringComparison.InvariantCultureIgnoreCase);
```

```
        };

        var recognizerInfo =
SpeechRecognitionEngine.InstalledRecognizers().Where(matchingFunc).FirstOrDefault();

        if (recognizerInfo == null)
            return;

        speechRecognitionEngine = new SpeechRecognitionEngine(recognizerInfo.Id);

        var gb = new GrammarBuilder { Culture = recognizerInfo.Culture };
        gb.Append(choices);

        var grammar = new Grammar(gb);

        speechRecognitionEngine.LoadGrammar(grammar);

        source.AutomaticGainControlEnabled = false;
        source.BeamAngleMode = BeamAngleMode.Adaptive;

        using (Stream sourceStream = source.Start())
        {
            speechRecognitionEngine.SetInputToAudioStream(sourceStream,
                new SpeechAudioFormatInfo(EncodingFormat.Pcm, 16000, 16, 1, 32000, 2, null));

            while (isRunning)
            {
                RecognitionResult result = speechRecognitionEngine.Recognize();
                if (result != null && OrderDetected != null && result.Confidence > 0.7)
                    OrderDetected(result.Text);
            }
        }
    }

    public void Stop()
    {
        isRunning = false;

        if (speechRecognitionEngine != null)
        {
            speechRecognitionEngine.Dispose();
        }
    }
}
```

Note that the code creates a thread that does not block the caller.

Using this class is quite simple, because you simply instantiate it and handle the event raised when a word is detected:

```
VoiceCommander voiceCommander = new VoiceCommander("record", "stop");
voiceCommander.OrderDetected += voiceCommander_OrderDetected;
```

You are now ready to start developing Kinect for Windows applications. You can even write unit tests for your code because, with the help of the record/replay system, the Kinect data is not just real-time.

PART III

Postures and gestures

CHAPTER 5

Capturing the context

Before you can start detecting gestures using the Kinect sensor with the help of the skeleton frame, it is important to understand the "discussion context."

This concept defines all the things that help you understand what a person wants to express. For example, when you are speaking with someone, you are also looking at that person and you can understand the gestures she makes while talking because of the discussion context. The gestures are just one part of a collection of information that also includes speech, body position, facial expressions, and so forth.

This point—that communication includes movement as well as words—is central when you are trying to detect gestures. Indeed, there are two categories of gestures: the desired and the undesired. The desired ones are those you produce willingly and that you want the system to detect. Undesired gestures are those that the system detects by mistake.

Capturing the "discussion context" helps you filter false gestures from the true gestures the system needs. Of course, for the system, it is not so much a matter of understanding what someone is doing or saying as it is using this context to get some hints about the will of the user.

In this chapter, you will write a set of tools to help you determine when a gesture is really a gesture.

The skeleton's stability

The first tool involves determining the stability of a skeleton. If a person is moving in front of the Kinect sensor, it is easy to conclude that he is not ready to give an order to your application. But to make this determination definitively, you must find the center of gravity of a skeleton (as shown in Figure 5-1). Fortunately, Kinect for Windows SDK provides this information for every detected skeleton.

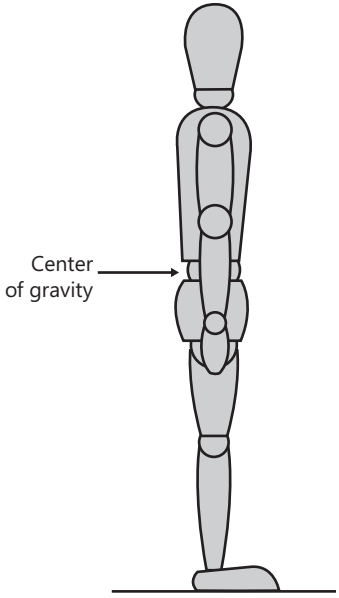

FIGURE 5-1 A human's average center of gravity.

To determine if a person is not moving, you have to record the last *n* positions (where *n* is 30 to 60, to cover half a second to a full second) and check to find out if the current position does not vary too much (given a specific threshold) from the average of the previous values.

The positions are stored using a simple struct called *Vector3*:

```
using System;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    [Serializable]
    public struct Vector3
    {
        public float X;
        public float Y;
        public float Z;

        public static Vector3 Zero
        {
            get
            {
                return new Vector3(0, 0, 0);
            }
        }

        public Vector3(float x, float y, float z)
        {
            X = x;
            Y = y;
```

```
            Z = z;
        }

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

        public static Vector3 operator -(Vector3 left, Vector3 right)
        {
            return new Vector3(left.X - right.X, left.Y - right.Y, left.Z - right.Z);
        }

        public static Vector3 operator +(Vector3 left, Vector3 right)
        {
            return new Vector3(left.X + right.X, left.Y + right.Y, left.Z + right.Z);
        }

        public static Vector3 operator *(Vector3 left, float value)
        {
            return new Vector3(left.X * value, left.Y * value, left.Z * value);
        }

        public static Vector3 operator *(float value, Vector3 left)
        {
            return left * value;
        }

        public static Vector3 operator /(Vector3 left, float value)
        {
            return new Vector3(left.X / value, left.Y / value, left.Z / value);
        }

        public static Vector3 ToVector3(SkeletonPoint vector)
        {
            return new Vector3(vector.X, vector.Y, vector.Z);
        }
    }
}
```

You now have to provide a class with a list of recorded positions and a public method to add a new position:

```
using System.Collections.Generic;
using System.Diagnostics;

namespace Kinect.Toolbox
{
    public class ContextTracker
    {
        readonly Dictionary<int, List<Vector3>> positions =
new Dictionary<int, List<Vector3>>();
        readonly int windowSize;
```

```
        public float Threshold { get; set; }

        public ContextTracker(int windowSize = 40, float threshold = 0.05f)
        {
            this.windowSize = windowSize;
            Threshold = threshold;
        }

        public void Add(Vector3 position, int trackingID)
        {
            if (!positions.ContainsKey(trackingID))
                positions.Add(trackingID, new List<Vector3>());

            positions[trackingID].Add(position);

            if (positions[trackingID].Count > windowSize)
                positions[trackingID].RemoveAt(0);
        }
    }
}
```

It is worth noting that the *Add* method awaits a position and a *trackingID*. The *trackingID* is provided by the skeletons (through the *Skeleton.TrackingId* property) and can be used to provide unique identifications for them.

Then you have to add a method to check the stability:

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

namespace Kinect.Toolbox
{
    public class ContextTracker
    {
        readonly Dictionary<int, List<Vector3>> positions =
new Dictionary<int, List<Vector3>>();
        readonly int windowSize;

        public float Threshold { get; set; }

        public ContextTracker(int windowSize = 40, float threshold = 0.05f)
        {
            this.windowSize = windowSize;
            Threshold = threshold;
        }

        public void Add(Vector3 position, int trackingID)
        {
            if (!positions.ContainsKey(trackingID))
                positions.Add(trackingID, new List<Vector3>());

            positions[trackingID].Add(position);

            if (positions[trackingID].Count > windowSize)
                positions[trackingID].RemoveAt(0);
```

```
        }

        public bool IsStable(int trackingID)
        {
            List<Vector3> currentPositions = positions[trackingID];
            if (currentPositions.Count != windowSize)
                return false;

            Vector3 current = currentPositions[currentPositions.Count - 1];

            Vector3 average = Vector3.Zero;

            for (int index = 0; index < currentPositions.Count - 2; index++)
            {
                average += currentPositions[index];
            }

            average /= currentPositions.Count - 1;

            if ((average - current).Length > Threshold)
                return false;


            return true;
        }
    }
}
```

In case the center of gravity is not ideal for you (because you are interested in tracking a specific part of the body), you should add another method to record position from a specific joint:

```
public void Add(Skeleton skeleton, JointType jointType)
{
    var trackingID = skeleton.TrackingId;
    var position = Vector3.ToVector3
(skeleton.Joints.Where(j => j.JointType == jointType).First().Position);

    Add(position, trackingID);
}
```

For now, you just have to check to see if the current skeleton is "stable" before trying to detect a gesture.

The skeleton's displacement speed

Another way to determine if the person in front of the Kinect sensor is making gestures that are useful to the system is to check the displacement speed of the skeleton. Staying totally stationary can be unnatural for a human body, and perfect stillness is hard to achieve. It is easier to stay *relatively* stationary, and so you can check the speed of the skeleton rather than its position.

To compute a speed, you need a position and a time, so the following class (*ContextPoint*) must be added to store these values:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Kinect.Toolbox
{
    public class ContextPoint
    {
        public DateTime Time
        {
            get;
            set;
        }

        public Vector3 Position
        {
            get;
            set;
        }
    }
}
```

You also need to update the *Add* method to record the current time along with the current position:

```
public void Add(Vector3 position, int trackingID)
 {
     if (!points.ContainsKey(trackingID))
     {
         points.Add(trackingID, new List<ContextPoint>());
     }

     points[trackingID].Add(new ContextPoint() { Position = position, Time = DateTime.Now });

     if (points[trackingID].Count > windowSize)
     {
         points[trackingID].RemoveAt(0);
     }
 }
```

You also must update the *IsStable* method to support the *ContextPoint* class:

```
public bool IsStable(int trackingID)
{
    List<ContextPoint> currentPoints = points[trackingID];
    if (currentPoints.Count != windowSize)
        return false;

    Vector3 current = currentPoints[currentPoints.Count - 1].Position;

    Vector3 average = Vector3.Zero;

    for (int index = 0; index < currentPoints.Count - 2; index++)
    {
```

```
            average += currentPoints[index].Position;
        }

        average /= currentPoints.Count - 1;

        if ((average - current).Length > Threshold)
            return false;

        return true;
    }
```

Now you have everything you need to determine the stability of the skeleton using its speed, and you can check to find out if the current (or the average) speed is under a given threshold, as shown in Figure 5-2.

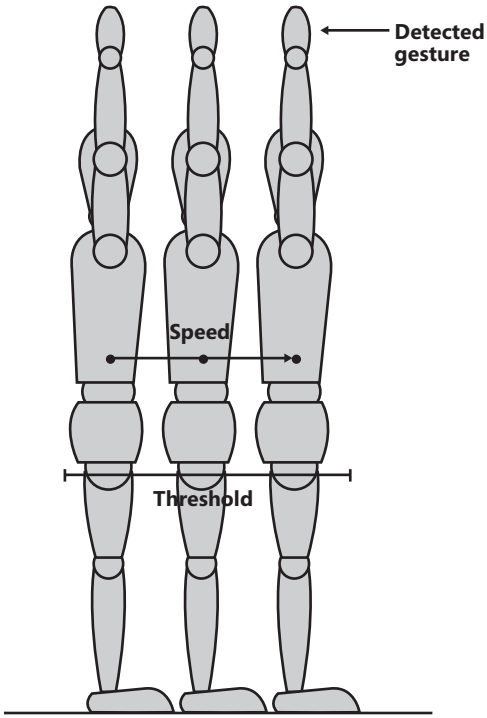

FIGURE 5-2 Detecting the stability of a skeleton based on the speed of its center of gravity in motion.

The code here is fairly simple:

```
public bool IsStableRelativeToCurrentSpeed(int trackingID)
{
    List<ContextPoint> currentPoints = points[trackingID];
    if (currentPoints.Count < 2)
        return false;
```

```
    Vector3 previousPosition = currentPoints[currentPoints.Count - 2].Position;
    Vector3 currentPosition = currentPoints[currentPoints.Count - 1].Position;

    DateTime previousTime = currentPoints[currentPoints.Count - 2].Time;
    DateTime currentTime = currentPoints[currentPoints.Count - 1].Time;

    var currentSpeed = (currentPosition - previousPosition).Length /
((currentTime - previousTime).TotalMilliseconds);

    if (currentSpeed > Threshold)
        return false;

    return true;
}

public bool IsStableRelativeToAverageSpeed(int trackingID)
{
    List<ContextPoint> currentPoints = points[trackingID];
    if (currentPoints.Count != windowSize)
        return false;

    Vector3 startPosition = currentPoints[0].Position;
    Vector3 currentPosition = currentPoints[currentPoints.Count - 1].Position;

    DateTime startTime = currentPoints[0].Time;
    DateTime currentTime = currentPoints[currentPoints.Count - 1].Time;

    var currentSpeed = (currentPosition - startPosition).Length /
((currentTime - startTime).TotalMilliseconds);

    if (currentSpeed > Threshold)
        return false;

    return true;
}
```

If you want to check the stability of a skeleton against its current speed, you can use *IsStableRelativeToCurrentSpeed,* and if you want to check against its average speed, you can use *IsStableRelativeToAverageSpeed*.

The skeleton's global orientation

You need one final tool before you can start capturing useful gestures. You must determine the direction of the user—the orientation of the skeleton is important because a gesture should only be detected when the user is in front of the sensor, focused on the sensor and facing toward it.

To detect that orientation, you can determine if the two shoulders of the skeleton are at the same distance (given a threshold) from the sensor, as shown in Figure 5-3.

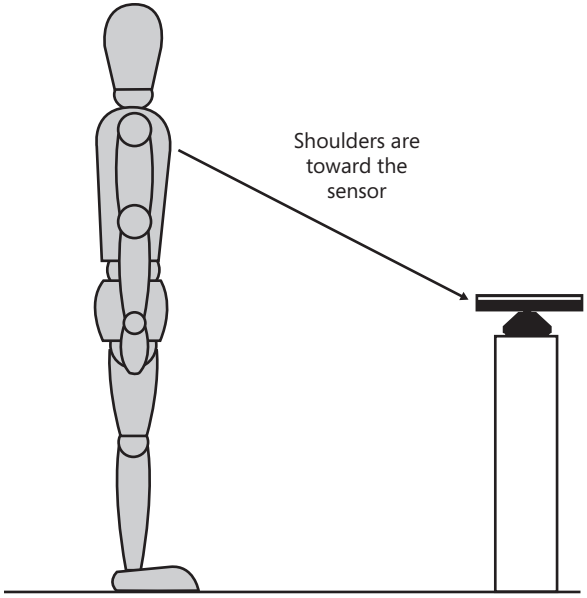

FIGURE 5-3 Determining if the attention of the user is focused on the Kinect sensor.

In this case, you have to check the position of the shoulders and compare the distance of each shoulder from the sensor (only on the z axis; you do not want to take into account the offset of the user on the x and y axes):

```
public bool IsShouldersTowardsSensor(Skeleton skeleton)
{
    var leftShoulderPosition = Vector3.ToVector3(skeleton.Joints.Where(j => j.JointType ==
JointType.ShoulderLeft).First().Position);
    var rightShoulderPosition = Vector3.ToVector3(skeleton.Joints.Where(j => j.JointType ==
JointType.ShoulderRight).First().Position);

    var leftDistance = leftShoulderPosition.Z;
    var rightDistance = rightShoulderPosition.Z;

    if (Math.Abs(leftDistance - rightDistance) > Threshold)
        return false;

    return true;
}
```

Complete *ContextTracker* tool code

The complete code for the *ContextTracker* tool is as follows:

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using Microsoft.Kinect;
```

```
using System;

namespace Kinect.Toolbox
{
    public class ContextTracker
    {
        readonly Dictionary<int, List<ContextPoint>> points =
new Dictionary<int, List<ContextPoint>>();
        readonly int windowSize;

        public float Threshold { get; set; }

        public ContextTracker(int windowSize = 40, float threshold = 0.05f)
        {
            this.windowSize = windowSize;
            Threshold = threshold;
        }

        public void Add(Vector3 position, int trackingID)
        {
            if (!points.ContainsKey(trackingID))
            {
                points.Add(trackingID, new List<ContextPoint>());
            }

            points[trackingID].Add(new ContextPoint()
{ Position = position, Time = DateTime.Now });

            if (points[trackingID].Count > windowSize)
            {
                points[trackingID].RemoveAt(0);
            }
        }

        public void Add(Skeleton skeleton, JointType jointType)
        {
            var trackingID = skeleton.TrackingId;
            var position = Vector3.ToVector3(skeleton.Joints.Where(j => j.JointType ==
jointType).First().Position);

            Add(position, trackingID);
        }

        public bool IsStable(int trackingID)
        {
            List<ContextPoint> currentPoints = points[trackingID];
            if (currentPoints.Count != windowSize)
                return false;

            Vector3 current = currentPoints[currentPoints.Count - 1].Position;

            Vector3 average = Vector3.Zero;

            for (int index = 0; index < currentPoints.Count - 2; index++)
            {
                average += currentPoints[index].Position;
            }
```

```
            average /= currentPoints.Count - 1;

            if ((average - current).Length > Threshold)
                return false;


            return true;
        }

        public bool IsStableRelativeToCurrentSpeed(int trackingID)
        {
            List<ContextPoint> currentPoints = points[trackingID];
            if (currentPoints.Count < 2)
                return false;

            Vector3 previousPosition = currentPoints[currentPoints.Count - 2].Position;
            Vector3 currentPosition = currentPoints[currentPoints.Count - 1].Position;

            DateTime previousTime = currentPoints[currentPoints.Count - 2].Time;
            DateTime currentTime = currentPoints[currentPoints.Count - 1].Time;

            var currentSpeed = (currentPosition - previousPosition).Length /
((currentTime - previousTime).TotalMilliseconds);

            if (currentSpeed > Threshold)
                return false;

            return true;
        }

        public bool IsStableRelativeToAverageSpeed(int trackingID)
        {
            List<ContextPoint> currentPoints = points[trackingID];
            if (currentPoints.Count != windowSize)
                return false;

            Vector3 startPosition = currentPoints[0].Position;
            Vector3 currentPosition = currentPoints[currentPoints.Count - 1].Position;

            DateTime startTime = currentPoints[0].Time;
            DateTime currentTime = currentPoints[currentPoints.Count - 1].Time;

            var currentSpeed = (currentPosition - startPosition).Length /
((currentTime - startTime).TotalMilliseconds);

            if (currentSpeed > Threshold)
                return false;

            return true;
        }

        public bool IsShouldersTowardsSensor(Skeleton skeleton)
        {
            var leftShoulderPosition = Vector3.ToVector3(skeleton.Joints.Where(j => j.JointType
== JointType.ShoulderLeft).First().Position);
            var rightShoulderPosition = Vector3.ToVector3(skeleton.Joints.Where(j => j.JointType
== JointType.ShoulderRight).First().Position);
```

```
            var leftDistance = leftShoulderPosition.Z;
            var rightDistance = rightShoulderPosition.Z;

            if (Math.Abs(leftDistance - rightDistance) > Threshold)
                return false;

            return true;
        }
    }
}
```

The *ContextTracker* tool will help you successfully detect gestures captured by the Kinect sensor by taking into consideration the "discussion context" of the user's movements.

Detecting the position of the skeleton's eyes

The Kinect for Windows SDK 1.5 introduced a new tool: the Kinect Toolkit. Using this toolkit, you can now detect points that make up the face of a skeleton. You can identify more than one hundred different points on the face of your user, and then you can use the points to create a three-dimensional (3D) mesh, as shown in Figure 5-4.

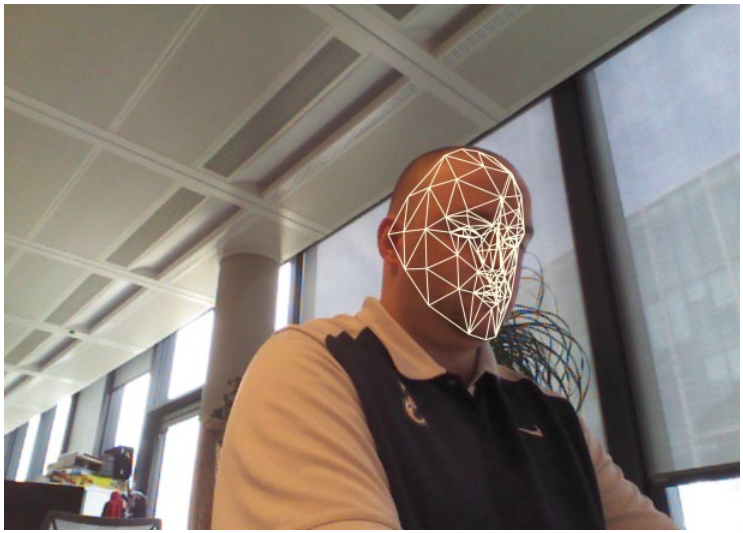

FIGURE 5-4 Drawing a 3D mesh on top of a user's face.

This information gives you a powerful tool that you can use to detect whether a user is currently looking at the sensor by determining the position of the user's eyes relative to the sensor. To determine whether the user is looking at the sensor, you compute the difference of the distance of each eye to the sensor. If this difference is under a defined epsilon, then the user is effectively looking at the sensor (similar to how you determine if both shoulders of a user are the same distance from the sensor).

You can now create a new class called *EyeTracker* that will provide an *IsLookingAtSensor* boolean property.

Note To use the face-tracking system in the Kinect Toolkit, you must add a reference to the *Microsoft.Kinect.Toolkit.FaceTracking* project or assembly (provided with Kinect for Windows SDK 1.5), and you must add a native dynamic link library (DLL) called *FaceTrackLib.dll* to your bin folder. This DLL is also provided by the SDK and can be added in your project with the Copy To Output Directory option activated.

The code for the *EyeTracker* class is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Kinect;
using Microsoft.Kinect.Toolkit.FaceTracking;

namespace Kinect.Toolbox
{
    public class EyeTracker : IDisposable
    {
        FaceTracker faceTracker;
        KinectSensor sensor;
        byte[] colors;
        short[] depths;
        float epsilon;

        public bool? IsLookingToSensor
        {
            get;
            private set;
        }

        public EyeTracker(KinectSensor sensor, float epsilon = 0.02f)
        {
            faceTracker = new FaceTracker(sensor);
            this.sensor = sensor;
            this.epsilon = epsilon;
        }

        public void Track(Skeleton skeleton)
        {
            // Colors
            if (colors == null)
            {
                colors = new byte[sensor.ColorStream.FramePixelDataLength];
            }

            var colorFrame = sensor.ColorStream.OpenNextFrame(0);
            if (colorFrame == null)
            {
```

```
                IsLookingToSensor = null;
                return;
            }

            colorFrame.CopyPixelDataTo(colors);

            // Depths
            if (depths == null)
            {
                depths = new short[sensor.DepthStream.FramePixelDataLength];
            }

            var depthFrame = sensor.DepthStream.OpenNextFrame(0);
            if (depthFrame == null)
            {
                IsLookingToSensor = null;
                return;
            }
            depthFrame.CopyPixelDataTo(depths);

            // Track
            var frame = faceTracker.Track(sensor.ColorStream.Format, colors, sensor.DepthStream.
Format, depths, skeleton);

            if (frame == null)
            {
                IsLookingToSensor = null;
                return;
            }
            var shape = frame.Get3DShape();

            var leftEyeZ = shape[FeaturePoint.AboveMidUpperLeftEyelid].Z;
            var rightEyeZ = shape[FeaturePoint.AboveMidUpperRightEyelid].Z;

            IsLookingToSensor = Math.Abs(leftEyeZ - rightEyeZ) <= epsilon;
        }

        public void Dispose()
        {
            faceTracker.Dispose();
        }
    }
}
```

As you can see, the face tracker needs the color and the depth stream values in order to create the shape of the face. After the shape is built, you can get the position of specific feature points (such as the eyelids, for instance).

This solution is quite accurate (you always look directly at the center of your interest), but it is also a highly CPU-intensive process, and this can lead to your user interface freezing up, so use it with caution.

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.

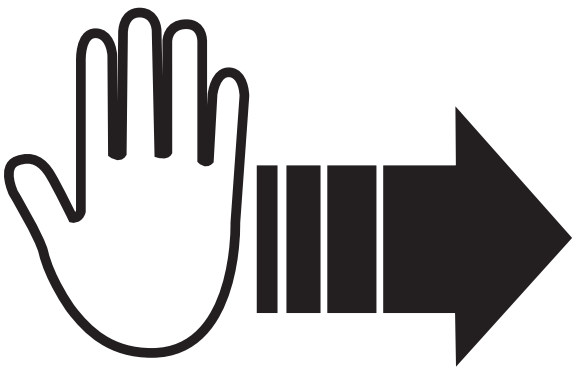

FIGURE 6-1 A gesture can be as simple as a hand swipe from left to right.

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)

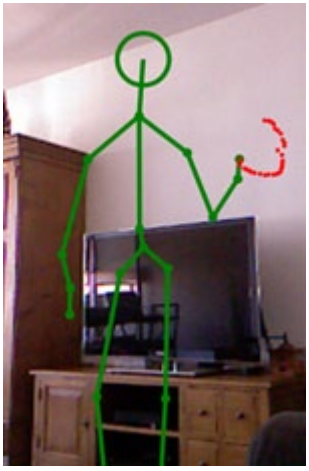

FIGURE 6-2 Drawing captured joint positions, shown in red (for readers of the print book, the captured joint positions are indicated by the semicircle of dots to the right of the skeleton).

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.

CHAPTER 7

Templated gestures and postures

In Chapter 6, "Algorithmic gestures and postures," you learned how to use algorithms to detect gestures and postures with the Kinect sensor. The main drawback of the algorithmic technique is that all gestures are not easily describable with constraints. In this chapter, you will learn a different approach, one that uses templates (previously recorded drawings) to identify postures and gestures.

Pattern matching gestures

In the real world, using algorithms to describe gestures can have limitations. For example, if you wanted to detect a circle gesture, you could spend days trying unsuccessfully trying to create an efficient algorithm that would incorporate all the constraints you would have to check for a circular movement. Sometimes a completely different approach is required—one with a more global vision. Pattern matching is another way of detecting gestures that can be used for even the most complicated movements.

Pattern matching involves the use of recorded drawings of gestures that serve as templates against which detected gestures can be compared. Using this method, gestures are represented by two-dimensional (2D) pictures in which each point is a position in time of a specific joint, as you can see in Figure 7-1.

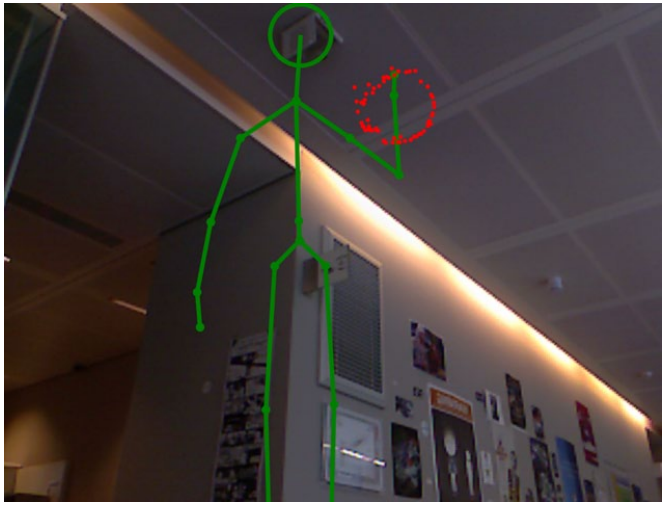

FIGURE 7-1 A 2D representation of a gesture.

Using these 2D drawings, you can develop pattern matching algorithms to determine if the gesture produced by a joint and detected by the Kinect sensor can be found in a list of recorded drawings of common gestures.

The main concept in pattern matching

The main concept in pattern matching involves having a database of recorded images produced by gestures. Using this database, you can compare the current drawing (made with the positions of the tracked joint) with all the recorded drawings (also called templates) in the database. A pattern matching algorithm (described later in this chapter) determines if one of the templates matches the current drawing, and if so, then the gesture is detected.

The first step is to write a database for your templates. For maximum efficiency, the database must include a lot of recorded drawings. You will, in a sense, be creating a learning machine that can "learn" new templates (that is, it will be able to store new drawings it encounters).

Machine learning is field of scientific research that has developed recently to create algorithms that give computers the ability to "learn" how to do certain tasks without specific programming. The research focuses on developing a way for computers to use empirical data (such as data from the sensor or information stored in a database) and to react and change based on the data they receive.

The learning machine that you create will be responsible for comparing a gesture presented to it with all of its templates to search for a match.

In the same way that the Kinect sensor works most efficiently with the most complete data, the more items the learning machine has, the better it will be at detecting and recognizing gestures. You will have to record many gestures made by many different people to increase the accuracy of the pattern matching.

Comparing the comparable

Before starting to write code to create your learning machine and matching algorithm, you have to *standardize* your data.

A gesture is a sequence of points. However, the coordinates of these points are related to the position of the sensor—every point is expressed in a coordinates system in which the sensor is at (0, 0)—and so you have to bring them together with a common reference that does not use the position of the sensor. If you don't do this, you will be forced to have the sensor always located at a defined position, a situation that is not workable.

It is far easier to compare drawings when they all have the same reference, the same number of points, and the same bounding box. To do this you have to

- Start with a complete gesture. A gesture is considered complete when it contains a specified number of points (as shown in Figure 7-2).

- Generate a new gesture with the defined number of points (as shown in Figure 7-3).
- Rotate the gesture so that the first point is at 0 degree (as shown in Figure 7-4).
- Rescale the gesture to a 1 × 1 reference graduation (as shown in Figure 7-5).
- Center the gesture to origin (as shown in Figure 7-6).

Note In Figures 7-2 through 7-6, the first point recorded for the gesture is shown in purple. For readers of the print book, this point appears in the upper left quadrant in Figures 7-2 and 7-3 and is located where the gesture drawing intersects the horizontal axis at left in Figures 7-4, 7-5, and 7-6.

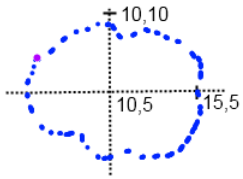

FIGURE 7-2 Create an initial gesture (with starting point in purple).

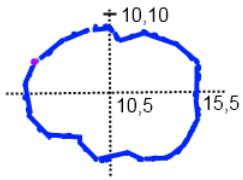

FIGURE 7-3 Normalize the gesture so it contains a fixed number of points.

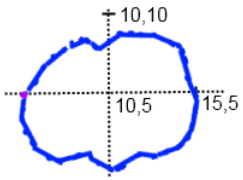

FIGURE 7-4 Rotate the drawing so that the first point is on the horizontal axis (at 0 degree).

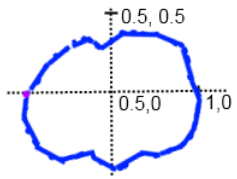

FIGURE 7-5 Rescale the gesture to a 1 × 1 reference graduation.

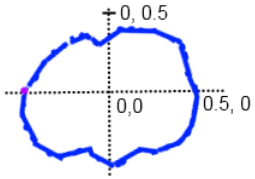

FIGURE 7-6 Center the gesture to the origin.

To be able to accomplish these operations successfully, you first must develop a method to normalize a gesture (defined by a list of *Vector2*) to a specific number of points.

Now let's work through all the steps required to complete the working code.

First, you need to know the length of a gesture (the sum of the distance between consecutive points). To do so, use the following extension method (more information about extension methods can be found at *http://msdn.microsoft.com/en-us/library/bb383977.aspx*):

```
// Get length of path
using System;
using System.Collections.Generic;
using System.Linq;
public static float Length(this List<Vector2> points)
{
    float length = 0;

    for (int i = 1; i < points.Count; i++)
    {
        length += (points[i - 1] - points[i]).Length;
    }

    return length;
}
```

So the code to normalize a gesture is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
static List<Vector2> ProjectListToDefinedCount(List<Vector2> positions, int n)
{
    List<Vector2> source = new List<Vector2>(positions);
    List<Vector2> destination = new List<Vector2>
                                    {
                                        source[0]
                                    };

    // define the average length of each segment
    float averageLength = positions.Length() / (n - 1);
    float currentDistance = 0;


    for (int index = 1; index < source.Count; index++)
    {
        Vector2 pt1 = source[index - 1];
```

```
        Vector2 pt2 = source[index];

        float distance = (pt1 - pt2).Length();
        // If the distance between the 2 points is greater than average length, we introduce a
new point
        if ((currentDistance + distance) >= averageLength)
        {
            Vector2 newPoint = pt1 +
((averageLength - currentDistance) / distance) * (pt2 - pt1);

            destination.Add(newPoint);
            source.Insert(index, newPoint);
            currentDistance = 0;
        }
        else
        {
            // merging points by ignoring it
            currentDistance += distance;
        }
    }

    if (destination.Count < n)
    {
        destination.Add(source[source.Count - 1]);
    }

    return destination;
}
```

This method computes the average length of a segment between two points, and for each segment it adds intermediate points if a segment is too long or removes points if a segment is too short.

For the next part, you need to know the center of a path:

```
// Get center of path
using System;
using System.Collections.Generic;
using System.Linq;
public static Vector2 Center(this List<Vector2> points)
{
    Vector2 result = points.Aggregate(Vector2.Zero, (current, point) => current + point);

    result /= points.Count;

    return result;
}
```

This is a simple procedure with Linq, isn't it?

More Info Find out more about Linq at *http://msdn.microsoft.com/en-us/library/bb397926.aspx*.

To rotate your normalized gestures, the following code is required:

```
using System;
using System.Collections.Generic;
using System.Linq;
// Rotate path by given angle
public static List<Vector2> Rotate(this List<Vector2> positions, float angle)
{
    List<Vector2> result = new List<Vector2>(positions.Count);
    Vector2 c = positions.Center();

    float cos = (float)Math.Cos(angle);
    float sin = (float)Math.Sin(angle);

    foreach (Vector2 p in positions)
    {
        float dx = p.X - c.X;
        float dy = p.Y - c.Y;

        Vector2 rotatePoint = Vector2.Zero;
        rotatePoint.X = dx * cos - dy * sin + c.X;
        rotatePoint.Y = dx * sin + dy * cos + c.Y;

        result.Add(rotatePoint);
    }
    return result;
}
```

Every point is rotated around the center by a given angle, and so the complete gesture rotates by the same angle.

To determine the required rotation angle, you must use the following method, which computes the angle between two points:

```
using System;
using System.Collections.Generic;
using System.Linq;
// A bit of trigonometry
public static float GetAngleBetween(Vector2 start, Vector2 end)
{
    if (start.X != end.X)
    {
        return (float)Math.Atan2(end.Y - start.Y, end.X - start.X);
    }

    if (end.Y > start.Y)
        return MathHelper.PiOver2;

    return -MathHelper.PiOver2;
}
```

When the gesture is correctly rotated, you can scale it to a 1 × 1 reference. To do so, you need an intermediate class to represent a bounding rectangle for the gesture:

```
namespace Kinect.Toolbox
{
    public struct Rectangle
```

```
    {
        public float X;
        public float Y;
        public float Width;
        public float Height;

        public Rectangle(float x, float y, float width, float height)
        {
            X = x;
            Y = y;
            Width = width;
            Height = height;
        }
    }
}
```

To compute a bounding rectangle from a list of points, use the following method:

```
// Compute bounding rectangle
public static Rectangle BoundingRectangle(this List<Vector2> points)
{
    float minX = points.Min(p => p.X);
    float maxX = points.Max(p => p.X);
    float minY = points.Min(p => p.Y);
    float maxY = points.Max(p => p.Y);

    return new Rectangle(minX, minY, maxX - minX, maxY - minY);
}
```

Then use the following code to scale the gesture:

```
using System;
using System.Collections.Generic;
using System.Linq;
// Scale path to 1x1
public static void ScaleToReferenceWorld(this List<Vector2> positions)
{
    Rectangle boundingRectangle = positions.BoundingRectangle();
    for (int i = 0; i < positions.Count; i++)
    {
        Vector2 position = positions[i];

        position.X *= (1.0f / boundingRectangle.Width);
        position.Y *= (1.0f / boundingRectangle.Height);

        positions[i] = position;
    }
}
```

Then you can finally center the gesture to the origin (0, 0):

```
using System;
using System.Collections.Generic;
using System.Linq;
// Translate path to origin (0, 0)
public static void CenterToOrigin(this List<Vector2> positions)
```

```
{
    Vector2 center = positions.Center();
    for (int i = 0; i < positions.Count; i++)
    {
        positions[i] -= center;
    }
}
```

For a given gesture, use the following code to prepare the gesture to be saved in the learning machine:

```
using System;
using System.Collections.Generic;
using System.Linq;
// Resample to required length then rotate to get first point at 0 radians, scale to 1x1 and
finally center the path to (0,0)
public static List<Vector2> Pack(List<Vector2> positions, int samplesCount)
{
    List<Vector2> locals = ProjectListToDefinedCount(positions, samplesCount);

    float angle = GetAngleBetween(locals.Center(), positions[0]);
    locals = locals.Rotate(-angle);

    locals.ScaleToReferenceWorld();
    locals.CenterToOrigin();

    return locals;
}
```

The golden section search

After you have saved gesture templates in your learning machine, there is one other relevant part of the template approach to detecting gestures that you need to complete—you must have a matching algorithm. This algorithm is used to compare the current gesture with all the gestures you have saved in the learning machine. The algorithm you use must be fast and it must be accurate, because you will need to call it many times per second.

On the Internet, you will find an interesting site created by Curtis F. Gerald and Patrick O. Wheatley at *http://homepages.math.uic.edu/~jan/mcs471/.* On this site, equation L-9 under "Solving Nonlinear Equations" is described as the golden section search algorithm. (You can find it at *http://homepages.math.uic.edu/~jan/mcs471/Lec9/lec9.html.)* This algorithm is exactly what you need—it provides a fast and accurate way to compare two normalized patterns.

The implementation of this algorithm is as follows:

```
using System;
using System.Collections.Generic;
static readonly float ReductionFactor = 0.5f * (-1 + (float)Math.Sqrt(5));
static readonly float Diagonal = (float)Math.Sqrt(2);

// a and b define the search interval [a, b] where a and b are angles
```

```
public static float Search(List<Vector2> current, List<Vector2> target,
float a, float b, float epsilon)
{
    float x1 = ReductionFactor * a + (1 - ReductionFactor) * b;
    List<Vector2> rotatedList = current.Rotate(x1);
    float fx1 = rotatedList.DistanceTo(target);

    float x2 = (1 - ReductionFactor) * a + ReductionFactor * b;
    rotatedList = current.Rotate(x2);
    float fx2 = rotatedList.DistanceTo(target);

    do
    {
        if (fx1 < fx2)
        {
            b = x2;
            x2 = x1;
            fx2 = fx1;
            x1 = ReductionFactor * a + (1 - ReductionFactor) * b;
            rotatedList = current.Rotate(x1);
            fx1 = rotatedList.DistanceTo(target);
        }
        else
        {
            a = x1;
            x1 = x2;
            fx1 = fx2;
            x2 = (1 - ReductionFactor) * a + ReductionFactor * b;
            rotatedList = current.Rotate(x2);
            fx2 = rotatedList.DistanceTo(target);
        }
    }
    while (Math.Abs(b - a) > epsilon);

    float min = Math.Min(fx1, fx2);

    return 1.0f - 2.0f * min / Diagonal;
}
```

The algorithm compares two patterns or templates, and if there is no match, then the first pattern is rotated slightly and the comparison process is launched again.

The epsilon value is used to define the precision of the algorithm. As a result, the method returns a confidence score (between 0 and 1) for the match.

Following is the code for all of the extension methods and the golden section search.:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Kinect.Toolbox
{
    public static class GoldenSectionExtensions
    {
        // Get length of path
```

```
        public static float Length(this List<Vector2> points)
        {
            float length = 0;

            for (int i = 1; i < points.Count; i++)
            {
                length += (points[i - 1] - points[i]).Length();
            }

            return length;
        }

        // Get center of path
        public static Vector2 Center(this List<Vector2> points)
        {
            Vector2 result = points.Aggregate(Vector2.Zero,
(current, point) => current + point);

            result /= points.Count;

            return result;
        }

        // Rotate path by given angle
        public static List<Vector2> Rotate(this List<Vector2> positions, float angle)
        {
            List<Vector2> result = new List<Vector2>(positions.Count);
            Vector2 c = positions.Center();

            float cos = (float)Math.Cos(angle);
            float sin = (float)Math.Sin(angle);

            foreach (Vector2 p in positions)
            {
                float dx = p.X - c.X;
                float dy = p.Y - c.Y;

                Vector2 rotatePoint = Vector2.Zero;
                rotatePoint.X = dx * cos - dy * sin + c.X;
                rotatePoint.Y = dx * sin + dy * cos + c.Y;

                result.Add(rotatePoint);
            }
            return result;
        }

        // Average distance between paths
        public static float DistanceTo(this List<Vector2> path1, List<Vector2> path2)
        {
            return path1.Select((t, i) => (t - path2[i]).Length()).Average();
        }

        // Compute bounding rectangle
        public static Rectangle BoundingRectangle(this List<Vector2> points)
        {
            float minX = points.Min(p => p.X);
            float maxX = points.Max(p => p.X);
```

```
            float minY = points.Min(p => p.Y);
            float maxY = points.Max(p => p.Y);

            return new Rectangle(minX, minY, maxX - minX, maxY - minY);
        }

        // Check bounding rectangle size
        public static bool IsLargeEnough(this List<Vector2> positions, float minSize)
        {
            Rectangle boundingRectangle = positions.BoundingRectangle();

            return boundingRectangle.Width > minSize && boundingRectangle.Height > minSize;
        }

        // Scale path to 1x1
        public static void ScaleToReferenceWorld(this List<Vector2> positions)
        {
            Rectangle boundingRectangle = positions.BoundingRectangle();
            for (int i = 0; i < positions.Count; i++)
            {
                Vector2 position = positions[i];

                position.X *= (1.0f / boundingRectangle.Width);
                position.Y *= (1.0f / boundingRectangle.Height);

                positions[i] = position;
            }
        }

        // Translate path to origin (0, 0)
        public static void CenterToOrigin(this List<Vector2> positions)
        {
            Vector2 center = positions.Center();
            for (int i = 0; i < positions.Count; i++)
            {
                positions[i] -= center;
            }
        }
    }
}

using System;
using System.Collections.Generic;

namespace Kinect.Toolbox.Gestures.Learning_Machine
{
    public static class GoldenSection
    {
        static readonly float ReductionFactor = 0.5f * (-1 + (float)Math.Sqrt(5));
        static readonly float Diagonal = (float)Math.Sqrt(2);

        public static float Search(List<Vector2> current, List<Vector2> target,
float a, float b, float epsilon)
        {
            float x1 = ReductionFactor * a + (1 - ReductionFactor) * b;
            List<Vector2> rotatedList = current.Rotate(x1);
            float fx1 = rotatedList.DistanceTo(target);
```

```
            float x2 = (1 - ReductionFactor) * a + ReductionFactor * b;
            rotatedList = current.Rotate(x2);
            float fx2 = rotatedList.DistanceTo(target);

            do
            {
                if (fx1 < fx2)
                {
                    b = x2;
                    x2 = x1;
                    fx2 = fx1;
                    x1 = ReductionFactor * a + (1 - ReductionFactor) * b;
                    rotatedList = current.Rotate(x1);
                    fx1 = rotatedList.DistanceTo(target);
                }
                else
                {
                    a = x1;
                    x1 = x2;
                    fx1 = fx2;
                    x2 = (1 - ReductionFactor) * a + ReductionFactor * b;
                    rotatedList = current.Rotate(x2);
                    fx2 = rotatedList.DistanceTo(target);
                }
            }
            while (Math.Abs(b - a) > epsilon);

            float min = Math.Min(fx1, fx2);

            return 1.0f - 2.0f * min / Diagonal;
        }

        static List<Vector2> ProjectListToDefinedCount(List<Vector2> positions, int n)
        {
            List<Vector2> source = new List<Vector2>(positions);
            List<Vector2> destination = new List<Vector2>
                                        {
                                            source[0]
                                        };

            // define the average length of each segment
            float averageLength = positions.Length() / (n - 1);
            float currentDistance = 0;


            for (int index = 1; index < source.Count; index++)
            {
                Vector2 pt1 = source[index - 1];
                Vector2 pt2 = source[index];

                float distance = (pt1 - pt2).Length();
                // If the distance between the 2 points is greater than average length, we
introduce a new point
                if ((currentDistance + distance) >= averageLength)
                {
                    Vector2 newPoint = pt1 +
```

```
((averageLength - currentDistance) / distance) * (pt2 - pt1);

                    destination.Add(newPoint);
                    source.Insert(index, newPoint);
                    currentDistance = 0;
                }
                else
                {
                    // merging points by ignoring it
                    currentDistance += distance;
                }
            }

            if (destination.Count < n)
            {
                destination.Add(source[source.Count - 1]);
            }

            return destination;
        }

        // A bit of trigonometry
        public static float GetAngleBetween(Vector2 start, Vector2 end)
        {
            if (start.X != end.X)
            {
                return (float)Math.Atan2(end.Y - start.Y, end.X - start.X);
            }

            if (end.Y > start.Y)
                return MathHelper.PiOver2;

            return -MathHelper.PiOver2;
        }

        // Resample to required length then rotate to get first point at 0 radians, scale to 1x1
and finally center the path to (0,0)
        public static List<Vector2> Pack(List<Vector2> positions, int samplesCount)
        {
            List<Vector2> locals = ProjectListToDefinedCount(positions, samplesCount);

            float angle = GetAngleBetween(locals.Center(), positions[0]);
            locals = locals.Rotate(-angle);

            locals.ScaleToReferenceWorld();
            locals.CenterToOrigin();

            return locals;
        }
    }
}
```

Creating a learning machine

You now have a good pattern-matching algorithm and a tool to normalize your gestures. You are ready to create the learning machine that will aggregate these two tools. The first step is to create a class to represent the learning machine database.

The *RecordedPath* class

As explained previously, the learning machine is a database filled with templates of recorded gestures. To represent a template, you can use the following class:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Windows;
using Kinect.Toolbox.Gestures.Learning_Machine; // For the GoldenSection methods
using System.Windows.Media.Imaging;
using System.Windows.Media;

namespace Kinect.Toolbox
{
    [Serializable]
    public class RecordedPath
    {
        List<Vector2> points;
        readonly int samplesCount;
        [NonSerialized]
        WriteableBitmap displayBitmap;

        public List<Vector2> Points
        {
            get { return points; }
            set { points = value; }
        }

        public WriteableBitmap DisplayBitmap
        {
            get
            {
                if (displayBitmap == null)
                {
                    displayBitmap =
new WriteableBitmap(200, 140, 96.0, 96.0, PixelFormats.Bgra32, null);

                    byte[] buffer =
new byte[displayBitmap.PixelWidth * displayBitmap.PixelHeight * 4];

                    foreach (Vector2 point in points)
                    {
                        int scaleX = (int)((point.X + 0.5f) * displayBitmap.PixelWidth);
                        int scaleY = (int)((point.Y + 0.5f) * displayBitmap.PixelHeight);

                        for (int x = scaleX - 2; x <= scaleX + 2; x++)
```

```
                        {
                            for (int y = scaleY - 2; y <= scaleY + 2; y++)
                            {
                                int clipX =
Math.Max(0, Math.Min(displayBitmap.PixelWidth - 1, x));
                                int clipY =
Math.Max(0, Math.Min(displayBitmap.PixelHeight - 1, y));
                                int index =
(clipX + clipY * displayBitmap.PixelWidth) * 4;

                                buffer[index] = 255;
                                buffer[index + 1] = 0;
                                buffer[index + 2] = 0;
                                buffer[index + 3] = 255;
                            }
                        }
                    }

                    displayBitmap.Lock();

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

                    displayBitmap.Unlock();
                }

                return displayBitmap;
            }
        }

        public RecordedPath(int samplesCount)
        {
            this.samplesCount = samplesCount;
            points = new List<Vector2>();
        }

        public void CloseAndPrepare()
        {
            points = GoldenSection.Pack(points, samplesCount);
        }

        public bool Match(List<Vector2> positions,
float threshold, float minimalScore, float minSize)
        {
            if (positions.Count < samplesCount)
                return false;

            if (!positions.IsLargeEnough(minSize))
                return false;

            List<Vector2> locals = GoldenSection.Pack(positions, samplesCount);

            float score =
```

```
GoldenSection.Search(locals, points, -MathHelper.PiOver4, MathHelper.PiOver4, threshold);

            Debug.WriteLine(score);

            return score > minimalScore;
        }
    }
}
```

The *RecordedPath* class provides the following information:

- A list of *Vector2* (the template itself)
- A tool to get a bitmap that represents the path of the template: *DisplayBitmap*
- A *Match* method to compare a list of *Vector2* with the current template
- A *CloseAndPrepare* method to normalize the current template (using your *Pack* method)

The *DisplayBitmap* property creates a *WriteableBitmap* and writes a pixel for every point of the template. This is mostly used for debugging purposes or to provide the user with visual feedback.

The *Match* method normalizes the input gesture and launches the golden section search between the internal template and the input normalized gesture.

Building the learning machine

Now that you have the *RecordedPath* class, you are ready to create your learning machine, which is essentially a simple list of *RecordedPath*. It must provide a *Match* method that calls the *Match* method of every *RecordedPath* it owns:

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using Kinect.Toolbox.Gestures.Learning_Machine; // for the RecordedPath class
using Microsoft.Xna.Framework;

namespace Kinect.Toolbox
{
    public class LearningMachine
    {
        readonly List<RecordedPath> paths;

        public LearningMachine(Stream kbStream)
        {
            if (kbStream == null || kbStream.Length == 0)
            {
                paths = new List<RecordedPath>();
                return;
            }

            BinaryFormatter formatter = new BinaryFormatter();
```

```
            paths = (List<RecordedPath>)formatter.Deserialize(kbStream);
        }

        public List<RecordedPath> Paths
        {
            get { return paths; }
        }

        public bool Match(List<Vector2> entries,
float threshold, float minimalScore, float minSize)
        {
            return Paths.Any(path => path.Match(entries, threshold, minimalScore, minSize));
        }

        public void Persist(Stream kbStream)
        {
            BinaryFormatter formatter = new BinaryFormatter();

            formatter.Serialize(kbStream, Paths);
        }

        public void AddPath(RecordedPath path)
        {
            path.CloseAndPrepare();
            Paths.Add(path);
        }
    }
}
```

As you can see, the learning machine also provides a way to serialize and deserialize its content using a *BinaryFormatter*.

The next step is to fill the learning machine you've created with many templates or patterns representing many different gestures!

For an accurate match, you will have to provide the learning machine you have created with as many templates as you can.

Detecting a gesture

Using your brand-new learning machine, you can start writing the code required to create the *TemplatedGestureDetector* class.

This class inherits from *GestureDetector* class (in the same way *AlgorithmicGestureDetector* does in Chapter 6):

```
using System.Linq;
using System.IO;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
```

```
    public class TemplatedGestureDetector : GestureDetector
    {
        public float Epsilon { get; set; }
        public float MinimalScore { get; set; }
        public float MinimalSize { get; set; }
        readonly LearningMachine learningMachine;
        RecordedPath path;
        readonly string gestureName;

        public bool IsRecordingPath
        {
            get { return path != null; }
        }

        public LearningMachine LearningMachine
        {
            get { return learningMachine; }
        }

        public TemplatedGestureDetector(string gestureName, Stream kbStream,
int windowSize = 60)
            : base(windowSize)
        {
            Epsilon = 0.035f;
            MinimalScore = 0.80f;
            MinimalSize = 0.1f;
            this.gestureName = gestureName;
            learningMachine = new LearningMachine(kbStream);
        }

        public override void Add(SkeletonPoint position, KinectSensor sensor)
        {
            base.Add(position, sensor);

            if (path != null)
            {
                path.Points.Add(new Vector2(position.X, position.Y));
            }
        }

        protected override void LookForGesture()
        {
            if (LearningMachine.Match(Entries.Select(e =>
new Vector2(e.Position.X, e.Position.Y)).ToList(), Epsilon, MinimalScore, MinimalSize))
                RaiseGestureDetected(gestureName);
        }

        public void StartRecordTemplate()
        {
            path = new RecordedPath(WindowSize);
        }
```

```
        public void EndRecordTemplate()
        {
            LearningMachine.AddPath(path);
            path = null;
        }

        public void SaveState(Stream kbStream)
        {
            LearningMachine.Persist(kbStream);
        }
    }
}
```

The *TemplatedGestureDetector* class overrides the *LookForGesture* method with a call to the *Match* method of its internal learning machine, passing it the list of saved joint positions. If the computed matching score is less than a specified value (80 percent here), then the *RaiseGestureDetected* is called.

Obviously, the class also provides a way to serialize (*SaveState*) and deserialize (*constructor*) itself. You can then simply save it to the disk or reload it with just a method call.

Finally, you can add a new path to the internal learning machine by first calling *StartRecordTemplate()* method. By doing that, any following calls to *Add* will save the joint position but will also add the position to a new recorded path. The path will be closed, packed, and integrated into the learning machine with a call to *EndRecordTemplat()e* method.

Detecting a posture

Good news—you can reuse the same technique for comparing a gesture with a recorded template to detect a posture. You can record the drawing created by recording a posture and integrate it in your learning machine.

Figure 7-7 shows the representation of a posture based on the position of the user.

FIGURE 7-7 Detecting a "T" posture.

You can then create a new *TemplatedGestureDetector* class that inherits from the *PostureDetector* class (in the same way *AlgorithmicPostureDetector* did in Chapter 6):

```
using System.IO;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class TemplatedPostureDetector : PostureDetector
    {
        public float Epsilon { get; set; }
        public float MinimalScore { get; set; }
        public float MinimalSize { get; set; }
        readonly LearningMachine learningMachine;
        readonly string postureName;

        public LearningMachine LearningMachine
        {
            get { return learningMachine; }
        }

        public TemplatedPostureDetector(string postureName, Stream kbStream) : base(4)
        {
            this.postureName = postureName;
            learningMachine = new LearningMachine(kbStream);

            MinimalScore = 0.95f;
            MinimalSize = 0.1f;
            Epsilon = 0.02f;
        }

        public override void TrackPostures(Skeleton skeleton)
        {
            if (LearningMachine.Match(skeleton.Joints.Select(j =>
new Vector2(j.Position.X, j.Position.Y)).ToList(), Epsilon, MinimalScore, MinimalSize))
                RaisePostureDetected(postureName);
        }

        public void AddTemplate(Skeleton skeleton)
        {
            RecordedPath recordedPath = new RecordedPath(skeleton.Joints.Count);

            recordedPath.Points.AddRange(skeleton.Joints.ToListOfVector2());

            LearningMachine.AddPath(recordedPath);
        }

        public void SaveState(Stream kbStream)
        {
            LearningMachine.Persist(kbStream);
        }
    }
}
```

This class is very similar to the *TemplatedGestureDetector* class. It provides a way to serialize and deserialize content, and you can easily add a new template to its learning machine (using the *AddTemplate* method).

The detection of a posture is similar to the detection of gestures: it requires a simple call to the *Match* method of the learning machine.

Going further with combined gestures

The last step in detecting really complex gestures involves recognizing combined gestures. That is, consider a very complex gesture as the *combination* of two or more basic gestures. For instance, in this way you can detect a complex gesture such as a hand describing a circle followed by a swipe to left.

To detect these kinds of combined gestures, add a new class named *CombinedGestureDetector* to your toolbox:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Kinect.Toolbox
{
    public abstract class CombinedGestureDetector : GestureDetector
    {
        List<GestureDetector> gestureDetectors = new List<GestureDetector>();

        public double Epsilon
        {
            get;
            private set;
        }

        public int GestureDetectorsCount
        {
            get
            {
                return gestureDetectors.Count;
            }
        }

        public CombinedGestureDetector(double epsilon = 1000)
        {
            Epsilon = epsilon;
        }

        public void Add(GestureDetector gestureDetector)
        {
            gestureDetector.OnGestureDetected += gestureDetector_OnGestureDetected;
            gestureDetectors.Add(gestureDetector);
        }
```

```
        public void Remove(GestureDetector gestureDetector)
        {
            gestureDetector.OnGestureDetected -= gestureDetector_OnGestureDetected;
            gestureDetectors.Remove(gestureDetector);
        }

        void gestureDetector_OnGestureDetected(string gesture)
        {
            CheckGestures(gesture);
        }

        protected abstract void CheckGestures(string gesture);

        protected override void LookForGesture()
        {
            // Do nothing
        }
    }
}
```

This class inherits from *GestureDetector* so that it will be considered as a *GestureDetector*; thus, you can chain *CombinedGestureDetector* (meaning that you can add a *CombinedGestureDetector* inside a *CombinedGestureDetector*).

For the most part, the code is built around the principle that every time a gesture is detected by a nested gesture detector, the class will call the abstract *CheckGestures* method to see if the combined gesture is complete.

Using this base class, you can add a new class named *ParallelCombinedGestureDetector* to handle multiple gestures in parallel (those detected at almost the same time):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Kinect.Toolbox
{
    public class ParallelCombinedGestureDetector : CombinedGestureDetector
    {
        DateTime? firstDetectedGestureTime;
        List<string> detectedGesturesName = new List<string>();

        public ParallelCombinedGestureDetector(double epsilon = 1000) : base(epsilon)
        {
        }

        protected override void CheckGestures(string gesture)
        {
            if (!firstDetectedGestureTime.HasValue ||
DateTime.Now.Subtract(firstDetectedGestureTime.Value).TotalMilliseconds >
Epsilon || detectedGesturesName.Contains(gesture))
            {
                firstDetectedGestureTime = DateTime.Now;
                detectedGesturesName.Clear();
            }
```

```
            detectedGesturesName.Add(gesture);

            if (detectedGesturesName.Count == GestureDetectorsCount)
            {
                RaiseGestureDetected(string.Join("&", detectedGesturesName));
                firstDetectedGestureTime = null;
            }
        }
    }
}
```

As you can see, the *CheckGestures* method checks to make sure every gesture is realized at almost the same time (given a specified epsilon value).

In a same way, you can add a new class called *SerialCombinedGestureDectector* to handle multiple gestures in a row:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Kinect.Toolbox
{
    public class SerialCombinedGestureDetector : CombinedGestureDetector
    {
        DateTime? previousGestureTime;
        List<string> detectedGesturesName = new List<string>();

        public SerialCombinedGestureDetector(double epsilon = 1000)
            : base(epsilon)
        {
        }

        protected override void CheckGestures(string gesture)
        {
            var currentTime = DateTime.Now;

            if (!previousGestureTime.HasValue || detectedGesturesName.Contains(gesture) ||
currentTime.Subtract(previousGestureTime.Value).TotalMilliseconds > Epsilon)
            {
                detectedGesturesName.Clear();
            }

            previousGestureTime = currentTime;

            detectedGesturesName.Add(gesture);
            if (detectedGesturesName.Count == GestureDetectorsCount)
            {
                RaiseGestureDetected(string.Join(">", detectedGesturesName));
                previousGestureTime = null;
            }
        }
    }
}
```

In this case, every gesture must be realized immediately after the previous one (given a specified *epsilon*). The name of the new gesture is produced by combining the names of the nested gestures separated by a ">" symbol, so you can deduce from the name the order in which the gestures were produced.

CHAPTER 8

Using gestures and postures in an application

With all the new tools you have assembled, you are now ready to create a real-world application that uses the Kinect sensor as a motion-sensing input device. The aim of this chapter is to help you integrate all the code you have written in previous chapters into a complete application.

The Gestures Viewer application

The application that you will create, called "Gestures Viewer," is intended to help you design and record the gestures and postures templates you will use in other applications. Figure 8-1 shows the main user interface (UI) of the application.

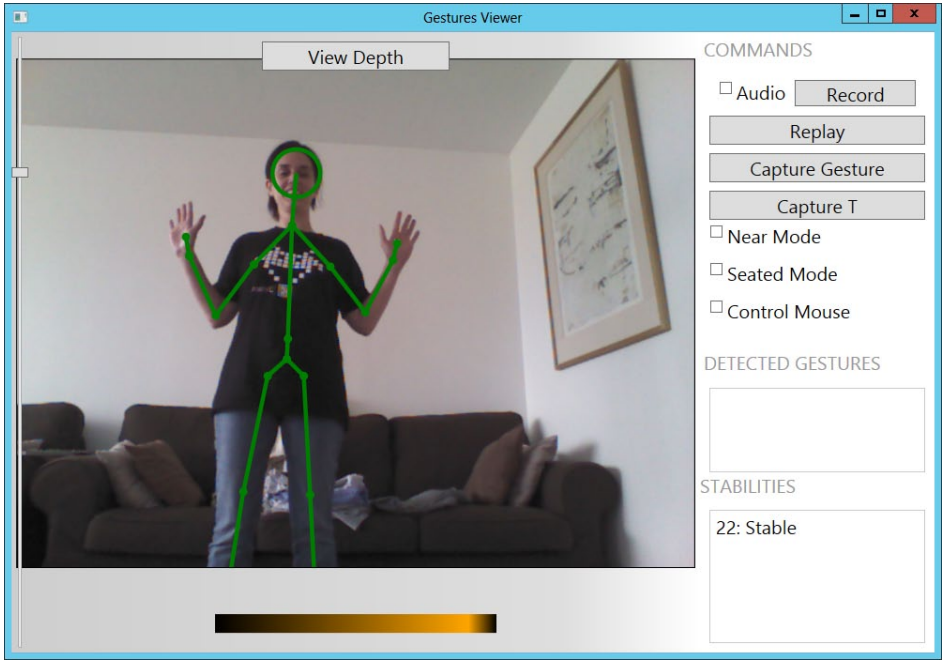

FIGURE 8-1 Using the Gestures Viewer application.

The application interface contains everything you need to record and detect new gestures and postures. It is also a good way to learn how to integrate Kinect with a Windows Presentation Foundation (WPF) application.

Gestures Viewer is built around a canvas that displays Kinect data (depth or color values). At the right of the UI, you can see most of the commands you will use to control the application; others appear above, below, and at the left of the display. These commands are shown labeled with numbers in Figure 8-2, and each command is explained in more detail in the following list.

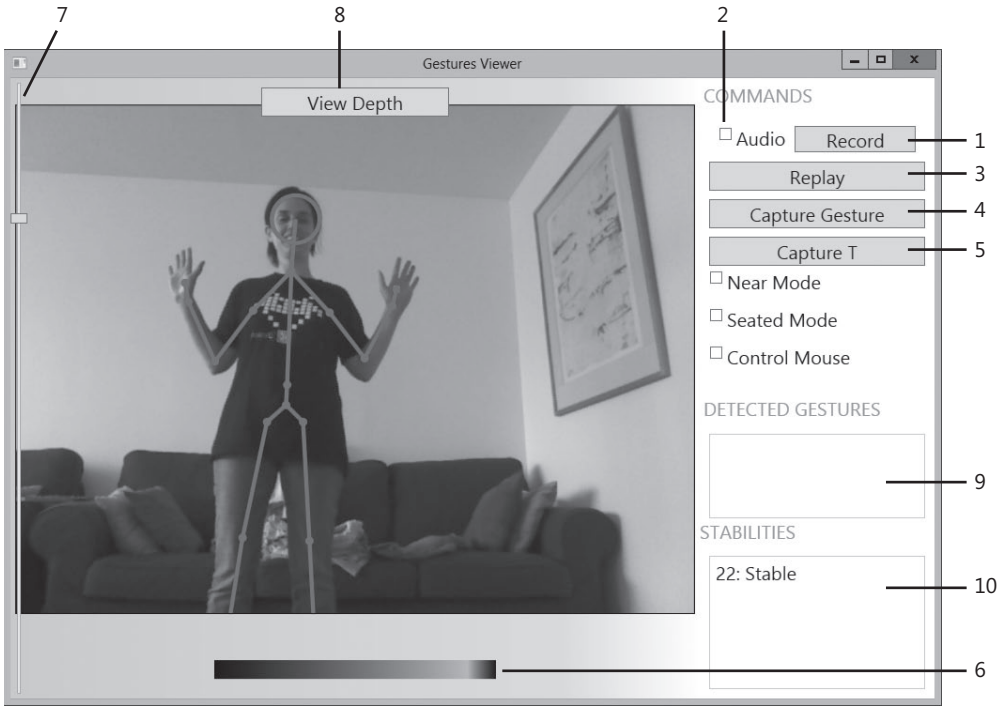

FIGURE 8-2 Commands available in the Gestures Viewer application.

1. *Record*: Use this button when you want to start recording a new Kinect session. Click the button again when the recording is complete.
2. *Audio*: Use this check box to control the recording of a session with your voice. A spoken "record" order will launch the recording, and a "stop" order will end it. This feature is useful when you are working alone and cannot be simultaneously in front of the sensor to record a session and in front of your computer to click the Record button.
3. *Replay:* Use this button to replay a previously recorded session.
4. *Capture Gesture*: Use this button when you want to start adding a new template to the gestures learning machine. Click the button again when the gesture template is complete.

5. *Capture T*: Use this button when you want to start adding a new template to the postures learning machine. Click this button again when the posture template is complete.
6. *Beam detection bar*: Use this bar to display the source of the sound using the beam detection code of the Kinect SDK.
7. *Elevation angle slider*: Use this slider to control the elevation of the Kinect sensor.
8. *Depth/Color button*: Use this button to switch between depth display and color display.
9. *Detected gestures*: This list displays the detected gestures.
10. *Stability list*: This list displays the stability state for each skeleton.

With all of these controls available, Gesture Viewer is a complete hub that allows you to create, refine, and test your gestures and postures.

Creating the user interface

Now that you know how the Gestures Viewer interface works, it's time to start developing! The first step is creating the user interface for the application. Gestures Viewer is a single-page XAML project based on a pair of canvases that are used to display depth and color values, accompanied by some buttons and list boxes used to control your Kinect code. Just create a new WPF project called *GesturesViewer* and paste the following code inside the XAML code of the MainWindow page:

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://schemas.
openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d" x:Class="GesturesViewer.MainWindow"
        Loaded="Window_Loaded"
        Closing="Window_Closing"
        WindowState="Normal"
        Title="Gestures Viewer" Height="700" Width="1000" MinHeight="700" MinWidth="1000"
MaxHeight="700" MaxWidth="1000">
    <Window.Resources>
        <Style TargetType="{x:Type TextBlock}">
            <Setter Property="FontFamily" Value="Segoe UI"/>
            <Setter Property="FontSize" Value="20"/>
            <Setter Property="Foreground" Value="#FF999999"/>
        </Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="FontFamily" Value="Segoe UI"/>
            <Setter Property="FontSize" Value="20"/>
            <Setter Property="Width" Value="200"/>
        </Style>
        <Style TargetType="{x:Type CheckBox}">
            <Setter Property="FontFamily" Value="Segoe UI"/>
            <Setter Property="FontSize" Value="20"/>
        </Style>
    </Window.Resources>
    <Grid>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition Width="250"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="200"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="150"/>
            <RowDefinition Height="30"/>
            <RowDefinition Height="150"/>
            <RowDefinition Height="*" MinHeight="30"/>
        </Grid.RowDefinitions>
        <Rectangle Grid.RowSpan="7">
            <Rectangle.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
                    <GradientStop Color="#FFCECECE" Offset="0"/>
                    <GradientStop Color="#FFDEDEDE" Offset="0.8"/>
                    <GradientStop Color="White" Offset="1"/>
                </LinearGradientBrush>
            </Rectangle.Fill>
        </Rectangle>
        <Viewbox Margin="5" Grid.RowSpan="6">
            <Grid Width="640" Height="480" ClipToBounds="True">
                <Image x:Name="kinectDisplay" Source="{Binding Bitmap}"/>
                <Canvas x:Name="kinectCanvas"/>
                <Canvas x:Name="gesturesCanvas"/>
                <Rectangle Stroke="Black" StrokeThickness="1"/>
            </Grid>
        </Viewbox>
        <TextBlock Text="COMMANDS" Grid.Column="1" Margin="4"/>
        <Grid Margin="10" Grid.Column="1" Grid.Row="1">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
                <CheckBox Content="Audio" x:Name="audioControl" VerticalAlignment="Center"
d:LayoutOverrides="Width"/>
                <Button Content="Record" x:Name="recordOption" Click="recordOption_Click"
VerticalAlignment="Center" Width="129" Margin="10,6,0,6"/>
            </StackPanel>
            <Button Content="Replay" x:Name="replayButton" Click="replayButton_Click"
VerticalAlignment="Center" Grid.Row="1" Width="Auto"/>
            <Button Content="Capture Gesture" x:Name="recordGesture" Click="recordGesture_Click"
VerticalAlignment="Center" Grid.Row="2" Width="Auto"/>
            <Button Content="Capture T" x:Name="recordT" Click="recordT_Click"
VerticalAlignment="Center" Grid.Row="3" Width="Auto"/>
        </Grid>
        <TextBlock Text="DETECTED GESTURES" Grid.Column="1" Grid.Row="2" Margin="4"/>
        <ListBox x:Name="detectedGestures" FontSize="20" Grid.Column="1" Margin="10,10,10,0"
Grid.Row="3"/>
        <StackPanel Grid.Row="6" Margin="212,0" HorizontalAlignment="Center"
VerticalAlignment="Center">
            <Rectangle x:Name="audioBeamAngle" Height="20" Width="300" Margin="5">
```

```
                <Rectangle.Fill>
                    <LinearGradientBrush StartPoint="0,0" EndPoint="1, 0">
                        <LinearGradientBrush.GradientStops>
                            <GradientStopCollection>
                                <GradientStop Offset="0" Color="Black"/>
                                <GradientStop Offset="{Binding BeamAngle}" Color="Orange"/>
                                <GradientStop Offset="1" Color="Black"/>
                            </GradientStopCollection>
                        </LinearGradientBrush.GradientStops>
                    </LinearGradientBrush>
                </Rectangle.Fill>
            </Rectangle>
            <StackPanel Orientation="Horizontal" />
        </StackPanel>
        <TextBlock Text="STABILITIES" Grid.Column="1" Grid.Row="4" />
        <ListBox x:Name="stabilitiesList" FontSize="20" Grid.Column="1" Grid.Row="5" Margin="10"
Grid.RowSpan="2">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Key}"/>
                        <TextBlock Text=": "/>
                        <TextBlock Text="{Binding Value}"/>
                    </StackPanel>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
        <Slider Minimum="{Binding ElevationMinimum}" Maximum="{Binding ElevationMaximum}"
Value="{Binding ElevationAngle, Mode=TwoWay}" x:Name="elevationSlider" Orientation="Vertical"
Grid.RowSpan="7"/>
        <Button Grid.RowSpan="2" Content="View Depth" Margin="267,10,267,0"
HorizontalAlignment="Center" VerticalAlignment="Top" Click="Button_Click" x:Name="viewButton"/>
    </Grid>
</Window>
```

The two canvases are contained inside a *Viewbox* to respect the 640 × 480 aspect ratio of Kinect data. Indeed, the *Viewbox* will stretch and scale its content in order to always keep the original resolution. It allows you to draw the skeletons on top of the Kinect streams (which are in a 640 × 480 format) because you always work in a virtual 640 × 480 panel (so you are in a 1 × 1 ratio between the canvases and your drawings). Except for the *Viewbox*, the code is standard XAML code, so you are now ready to move on to the next step in creating the Gestures Viewer application. Please note that for now, the code will not compile because you have to add some events in the code behind file.

Initializing the application

The initialization of your application must make a connection to a Kinect sensor and create all of the following required objects:

- A *SwipeGestureDetector* and a *TemplatedGestureDetector* for gestures (described in Chapter 6, "Algorithmic gestures and postures" and Chapter 7, "Templated gestures and postures")
- A *ColorSreamManager* and a *DepthStreamManager* for the display (described in Chapter 3, "Displaying Kinect data")

- An *AudioStreamManager* for the beam display (described in Chapter 3)
- A *SkeletonDisplayManager* for displaying skeletons (described in Chapter 3)
- A *ContextTracker* for the stability (described in Chapter 5, "Capturing the context")
- An *AlgorithmicPostureDetector* and a *TemplatedPostureDetector* for the postures (described in Chapter 6 and Chapter 7)
- A *KinectRecorder*, a *KinectReplay,* and some associated variables to record and replay sessions (described in Chapter 4, "Recording and replaying a Kinect session")
- A *BindableNUICamera* to control the angle of the sensor

Note You will create the *BindableNUICamera* class later in this chapter.

- An array of detected skeletons
- A *VoiceCommander* to control the application with your voice (described in Chapter 4)
- Two paths for loading and saving learning machines for gestures and postures

The declaration part of your application is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using Kinect.Toolbox;
using Kinect.Toolbox.Record;
using System.IO;
using Microsoft.Kinect;
using Microsoft.Win32;
using Kinect.Toolbox.Voice;

namespace GesturesViewer
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow
    {
        KinectSensor kinectSensor;

        SwipeGestureDetector swipeGestureRecognizer;
        TemplatedGestureDetector circleGestureRecognizer;
        readonly ColorStreamManager colorManager = new ColorStreamManager();
        readonly DepthStreamManager depthManager = new DepthStreamManager();
        AudioStreamManager audioManager;
        SkeletonDisplayManager skeletonDisplayManager;
        readonly ContextTracker contextTracker = new ContextTracker();
        readonly AlgorithmicPostureDetector algorithmicPostureRecognizer =
new AlgorithmicPostureDetector();
```

```
        TemplatedPostureDetector templatePostureDetector;
        private bool recordNextFrameForPosture;
        bool displayDepth;

        string circleKBPath;
        string letterT_KBPath;

        KinectRecorder recorder;
        KinectReplay replay;

        BindableNUICamera nuiCamera;

        private Skeleton[] skeletons;

        VoiceCommander voiceCommander;

        public MainWindow()
        {
            InitializeComponent();
        }
```

Within the load event, you have to detect the presence of a Kinect sensor and then launch the initialization code or wait for a Kinect sensor to be available:

```
void Kinects_StatusChanged(object sender, StatusChangedEventArgs e)
{
    switch (e.Status)
    {
        case KinectStatus.Connected:
            if (kinectSensor == null)
            {
                kinectSensor = e.Sensor;
                Initialize();
            }
            break;
        case KinectStatus.Disconnected:
            if (kinectSensor == e.Sensor)
            {
                Clean();
                MessageBox.Show("Kinect was disconnected");
            }
            break;
        case KinectStatus.NotReady:
            break;
        case KinectStatus.NotPowered:
            if (kinectSensor == e.Sensor)
            {
                Clean();
                MessageBox.Show("Kinect is no longer powered");
            }
            break;
        default:
            MessageBox.Show("Unhandled Status: " + e.Status);
            break;
    }
}
```

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            circleKBPath = Path.Combine(Environment.CurrentDirectory, @"data\circleKB.save");
            letterT_KBPath = Path.Combine(Environment.CurrentDirectory, @"data\t_KB.save");

            try
            {
                //listen to any status change for Kinects
                KinectSensor.KinectSensors.StatusChanged += Kinects_StatusChanged;

                //loop through all the Kinects attached to this PC,
and start the first that is connected without an error.
                foreach (KinectSensor kinect in KinectSensor.KinectSensors)
                {
                    if (kinect.Status == KinectStatus.Connected)
                    {
                        kinectSensor = kinect;
                        break;
                    }
                }

                if (KinectSensor.KinectSensors.Count == 0)
                    MessageBox.Show("No Kinect found");
                else
                    Initialize();

            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        private void Initialize()
        {
            if (kinectSensor == null)
                return;

            audioManager = new AudioStreamManager(kinectSensor.AudioSource);
            audioBeamAngle.DataContext = audioManager;

            kinectSensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
            kinectSensor.ColorFrameReady += kinectRuntime_ColorFrameReady;

            kinectSensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
            kinectSensor.DepthFrameReady += kinectSensor_DepthFrameReady;

            kinectSensor.SkeletonStream.Enable(new TransformSmoothParameters
                                                {
                                                    Smoothing = 0.5f,
                                                    Correction = 0.5f,
                                                    Prediction = 0.5f,
                                                    JitterRadius = 0.05f,
                                                    MaxDeviationRadius = 0.04f
                                                });
            kinectSensor.SkeletonFrameReady += kinectRuntime_SkeletonFrameReady;
```

```
            swipeGestureRecognizer = new SwipeGestureDetector();
            swipeGestureRecognizer.OnGestureDetected += OnGestureDetected;

            skeletonDisplayManager = new SkeletonDisplayManager(kinectSensor, kinectCanvas);

            kinectSensor.Start();

            LoadGestureDetector();
            LoadLetterTPostureDetector();

            nuiCamera = new BindableNUICamera(kinectSensor);

            elevationSlider.DataContext = nuiCamera;

            voiceCommander = new VoiceCommander("record", "stop");
            voiceCommander.OrderDetected += voiceCommander_OrderDetected;

            StartVoiceCommander();

            kinectDisplay.DataContext = colorManager;
        }
```

As you can see, the *Initialize* method is called when a Kinect sensor is connected. This method calls many methods described later in this chapter.

The following code is provided to initialize the *TemplatedGestureDetector*:

```
void LoadGestureDetector()
        {
            using (Stream recordStream = File.Open(circleKBPath, FileMode.OpenOrCreate))
            {
                circleGestureRecognizer = new TemplatedGestureDetector("Circle", recordStream);
                circleGestureRecognizer.DisplayCanvas = gesturesCanvas;
                circleGestureRecognizer.OnGestureDetected += OnGestureDetected;
            }
        }
```

When the detector detects a gesture, it raises the following event:

```
        void OnGestureDetected(string gesture)
        {
            int pos = detectedGestures.Items.Add
(string.Format("{0} : {1}", gesture, DateTime.Now));

            detectedGestures.SelectedIndex = pos;
        }
```

The *TemplatedPostureDetector* is initialized the same way:

```
        void LoadLetterTPostureDetector()
        {
            using (Stream recordStream = File.Open(letterT_KBPath, FileMode.OpenOrCreate))
            {
                templatePostureDetector = new TemplatedPostureDetector("T", recordStream);
```

```
                templatePostureDetector.PostureDetected +=
templatePostureDetector_PostureDetected;
            }
        }
```

And when a posture is detected, the following code is called:

```
        void templatePostureDetector_PostureDetected(string posture)
        {
            MessageBox.Show("Give me a......." + posture);
        }
```

Displaying Kinect data

The *Depth/ColorStreamManager* classes are used to display Kinect data on the canvases. Every kind of data is processed by a *Depth/ColorStreamManager* object and a dedicated event (which is raised when data is available):

```
    // Depth data
    void kinectSensor_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)
        {
            if (replay != null && !replay.IsFinished)
                return;

            using (var frame = e.OpenDepthImageFrame())
            {
                if (frame == null)
                    return;

                if (recorder != null && ((recorder.Options & KinectRecordOptions.Depth) != 0))
                {
                    recorder.Record(frame);
                }

                if (!displayDepth)
                    return;

                depthManager.Update(frame);
            }
        }

        // Color data
        void kinectRuntime_ColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)
        {
            if (replay != null && !replay.IsFinished)
                return;

            using (var frame = e.OpenColorImageFrame())
            {
                if (frame == null)
                    return;

                if (recorder != null && ((recorder.Options & KinectRecordOptions.Color) != 0))
                {
                    recorder.Record(frame);
                }
```

```
                if (displayDepth)
                    return;

                colorManager.Update(frame);
            }
        }

        // Skeleton data
        void kinectRuntime_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
        {
            if (replay != null && !replay.IsFinished)
                return;

            using (SkeletonFrame frame = e.OpenSkeletonFrame())
            {
                if (frame == null)
                    return;

                if (recorder != null &&
((recorder.Options & KinectRecordOptions.Skeletons) != 0))
                    recorder.Record(frame);

                Tools.GetSkeletons(frame, ref skeletons);

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

                ProcessFrame(frame);
            }
        }
```

Notice that every event is responsible for controlling the correct *Depth/ColorStreamManager*. It is also responsible for calling the recorder class if the session record is activated (as you will see later in this chapter).

The *kinectRuntime_SkeletonFrameReady* is also responsible for calling the *ProcessFrame* method (which you will see later); that method is used to detect gestures and postures.

When the user clicks the View Depth/View Color button, the right canvas is displayed:

```
private void Button_Click(object sender, RoutedEventArgs e)
        {
            displayDepth = !displayDepth;

            if (displayDepth)
            {
                viewButton.Content = "View Color";
                kinectDisplay.DataContext = depthManager;
            }
            else
            {
                viewButton.Content = "View Depth";
                kinectDisplay.DataContext = colorManager;
            }
        }
```

Controlling the angle of the Kinect sensor

To control the elevation angle of the Kinect sensor, you can use the following *BindableNUICamera* class:

```
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class BindableNUICamera : Notifier
    {
        readonly KinectSensor nuiCamera;
        public int ElevationAngle
        {
            get { return nuiCamera.ElevationAngle; }
            set
            {
                if (nuiCamera.ElevationAngle == value)
                    return;

                if (value > ElevationMaximum)
                    value = ElevationMaximum;

                if (value < ElevationMinimum)
                    value = ElevationMinimum;

                nuiCamera.ElevationAngle = value;
RaisePropertyChanged(() => ElevationAngle);
            }
        }

        public int ElevationMaximum
        {
            get { return nuiCamera.MaxElevationAngle; }
        }


        public int ElevationMinimum
        {
            get { return nuiCamera.MinElevationAngle; }
        }

        public BindableNUICamera(KinectSensor nuiCamera)
        {
            this.nuiCamera = nuiCamera;
        }
    }
}
```

As you saw previously in the *MainWindow* constructor, this class is bound to the left slider:

```
nuiCamera = new BindableNUICamera(kinectSensor);

elevationSlider.DataContext = nuiCamera;
```

Detecting gestures and postures with Gestures Viewer

The *ProcessFrame* method is called when a skeleton frame is ready. It is then responsible for feeding the gesture and posture detector classes after checking the stability with the context tracker:

```
void ProcessFrame(ReplaySkeletonFrame frame)
{
    Dictionary<int, string> stabilities = new Dictionary<int, string>();
    foreach (var skeleton in frame.Skeletons)
    {
        if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
            continue;

       // Stability?
        contextTracker.Add(skeleton.Position.ToVector3(), skeleton.TrackingId);
        stabilities.Add(skeleton.TrackingId,
contextTracker.IsStableRelativeToCurrentSpeed(skeleton.TrackingId) ? "Stable" : "Non stable");
        if (!contextTracker.IsStableRelativeToCurrentSpeed(skeleton.TrackingId))
            continue;

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

            if (joint.JointType == JointType.HandRight)
            {
                swipeGestureRecognizer.Add(joint.Position, kinectSensor);
                circleGestureRecognizer.Add(joint.Position, kinectSensor);
            }
        }

        algorithmicPostureRecognizer.TrackPostures(skeleton);
        templatePostureDetector.TrackPostures(skeleton);

        if (recordNextFrameForPosture)
        {
            templatePostureDetector.AddTemplate(skeleton);
            recordNextFrameForPosture = false;
        }
    }

    skeletonDisplayManager.Draw(frame);

    stabilitiesList.ItemsSource = stabilities;
}
```

Recording and replaying a session

The recording part of the application is dispatched inside the Kinect data events, but it also uses the following code:

```
using System.IO;
using System.Windows;
using Kinect.Toolbox.Record;
using Microsoft.Win32;

namespace GesturesViewer
{
    partial class MainWindow
    {
        private void recordOption_Click(object sender, RoutedEventArgs e)
        {
            if (recorder != null)
            {
                StopRecord();
            }

            SaveFileDialog saveFileDialog =
new SaveFileDialog { Title = "Select filename", Filter = "Replay files|*.replay" };

            if (saveFileDialog.ShowDialog() == true)
            {
                DirectRecord(saveFileDialog.FileName);
            }
        }

        void DirectRecord(string targetFileName)
        {
            Stream recordStream = File.Create(targetFileName);
            recorder = new KinectRecorder(KinectRecordOptions.Skeletons |
KinectRecordOptions.Color | KinectRecordOptions.Depth, recordStream);
            recordOption.Content = "Stop Recording";
        }

        void StopRecord()
        {
            if (recorder != null)
            {
                recorder.Stop();
                recorder = null;
                recordOption.Content = "Record";
                return;
            }
        }
    }
}
```

Here the main job of the code is getting a file with a *SaveFileDialog* so that it can retrieve a stream for the *KinectRecorder* class.

The replay part of the application does the same thing—it retrieves a stream for the *KinectReplay* class and uses three intermediate events to reproduce the behavior of the original Kinect events:

```
        private void replayButton_Click(object sender, RoutedEventArgs e)
        {
            OpenFileDialog openFileDialog = new OpenFileDialog
{ Title = "Select filename", Filter = "Replay files|*.replay" };
```

```
            if (openFileDialog.ShowDialog() == true)
            {
                if (replay != null)
                {
                    replay.SkeletonFrameReady -= replay_SkeletonFrameReady;
                    replay.ColorImageFrameReady -= replay_ColorImageFrameReady;
                    replay.Stop();
                }
                Stream recordStream = File.OpenRead(openFileDialog.FileName);

                replay = new KinectReplay(recordStream);

                replay.SkeletonFrameReady += replay_SkeletonFrameReady;
                replay.ColorImageFrameReady += replay_ColorImageFrameReady;
                replay.DepthImageFrameReady += replay_DepthImageFrameReady;

                replay.Start();
            }
        }

        void replay_DepthImageFrameReady(object sender, ReplayDepthImageFrameReadyEventArgs e)
        {
            if (!displayDepth)
                return;

            depthManager.Update(e.DepthImageFrame);
        }

        void replay_ColorImageFrameReady(object sender, ReplayColorImageFrameReadyEventArgs e)
        {
            if (displayDepth)
                return;

            colorManager.Update(e.ColorImageFrame);
        }

        void replay_SkeletonFrameReady(object sender, ReplaySkeletonFrameReadyEventArgs e)
        {
            ProcessFrame(e.SkeletonFrame);
        }
```

Obviously, each *replay Depth/ColorFrameReady* has the same behavior as *kinectSensor_Depth/ColorFrameReady*.

Recording new gestures and postures

The *ProcessFrame* method is also in charge of recording gestures and postures with the following methods.

This code records gestures:

```
using System;
using System.IO;
using System.Windows;
using System.Windows.Media;
```

```
using Kinect.Toolbox;
using Microsoft.Kinect;

namespace GesturesViewer
{
    partial class MainWindow
    {
        private void recordGesture_Click(object sender, RoutedEventArgs e)
        {
            if (circleGestureRecognizer.IsRecordingPath)
            {
                circleGestureRecognizer.EndRecordTemplate();
                recordGesture.Content = "Record Gesture";
                return;
            }

            circleGestureRecognizer.StartRecordTemplate();
            recordGesture.Content = "Stop Recording";
        }
    }
}
```

The *recordGesture* button calls the *StartRecordingTemplate* method of the *TemplatedGestureRecognizer* on first click and then calls the *EndRecordTemplate* method to finalize the recorded template.

This code records postures:

```
using System.IO;
using System.Windows;
using Kinect.Toolbox;

namespace GesturesViewer
{
    partial class MainWindow
    {
        private void recordT_Click(object sender, RoutedEventArgs e)
        {
           recordNextFrameForPosture = true;
        }
    }
}
```

The recording of a posture is based on the *recordNextFrameForPosture* boolean. When it is true, the *ProcessFrame* method record the current posture in the *TemplatedPostureDectector* object:

```
        if (recordNextFrameForPosture)
        {
            templatePostureDetector.AddTemplate(skeleton);
            recordNextFrameForPosture = false;
        }
```

Commanding Gestures Viewer with your voice

The *VoiceCommander* class is used to track the words "record" and "stop" to control the session recording:

```
using System;
using System.IO;

namespace GesturesViewer
{
    partial class MainWindow
    {
        void StartVoiceCommander()
        {
            voiceCommander.Start(kinectSensor);
        }

        void voiceCommander_OrderDetected(string order)
        {
            Dispatcher.Invoke(new Action(() =>
            {
                if (audioControl.IsChecked == false)
                    return;

                switch (order)
                {
                    case "record":
                        DirectRecord
(Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Desktop), "kinectRecord" +
Guid.NewGuid() + ".replay"));
                        break;
                    case "stop":
                        StopRecord();
                        break;
                }
            }));
        }
    }
}
```

Because the user cannot select a file when using the *VoiceCommander*, the application creates a temporary file called *kinectRecord* with a suffix composed of a random globally unique identifier (GUID). The file is saved on the user's desktop.

Using the beam angle

The beam angle is simply displayed by the *AudioStreamManager*:

```
audioManager = new AudioStreamManager(kinectSensor.AudioSource);
audioBeamAngle.DataContext = audioManager;
```

Cleaning resources

Finally, it is important to dispose of all the resources you used. The following code accomplishes this clean up:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    Clean();
}

void CloseGestureDetector()
{
    if (circleGestureRecognizer == null)
        return;

    using (Stream recordStream = File.Create(circleKBPath))
    {
        circleGestureRecognizer.SaveState(recordStream);
    }
    circleGestureRecognizer.OnGestureDetected -= OnGestureDetected;
}


void ClosePostureDetector()
{
    if (templatePostureDetector == null)
        return;

    using (Stream recordStream = File.Create(letterT_KBPath))
    {
        templatePostureDetector.SaveState(recordStream);
    }
    templatePostureDetector.PostureDetected -= templatePostureDetector_PostureDetected;
}

private void Clean()
{
    if (swipeGestureRecognizer != null)
    {
        swipeGestureRecognizer.OnGestureDetected -= OnGestureDetected;
    }

    if (audioManager != null)
    {
        audioManager.Dispose();
        audioManager = null;
    }

    CloseGestureDetector();

    ClosePostureDetector();

    if (voiceCommander != null)
    {
        voiceCommander.OrderDetected -= voiceCommander_OrderDetected;
```

```
            voiceCommander.Stop();
            voiceCommander = null;
        }

        if (recorder != null)
        {
            recorder.Stop();
            recorder = null;
        }

        if (kinectSensor != null)
        {
            kinectSensor.ColorFrameReady -= kinectRuntime_ColorFrameReady;
            kinectSensor.SkeletonFrameReady -= kinectRuntime_SkeletonFrameReady;
            kinectSensor.ColorFrameReady -= kinectRuntime_ColorFrameReady;
            kinectSensor.Stop();
            kinectSensor = null;
        }
    }
```

Now that you have created and used the Gestures Viewer application, you have a good idea of how to use the tools you developed in previous chapters. The application can also help you fine-tune and fill your learning machines with templates of gestures and postures for the Kinect sensor to detect and recognize for more complex applications.

PART IV

Creating a user interface for Kinect

CHAPTER 9

You are the mouse!

Kinect is clearly a new way to communicate with the computer. The world of applications quickly took advantage of the versatility of the mouse when it was introduced in the ancient ages of computer science, and applications now must evolve to take into account a new way of obtaining input data through the Kinect sensor.

Before the mouse was invented, user interfaces were all based on the keyboard. Users had to type directions and data, and they had to use the Tab key to move from one field to another. Screens were based on text boxes and text blocks.

Then the mouse appeared, and user interfaces quickly started to change and adapt to a new way of providing input to the computer. Graphical user interfaces (GUIs) appeared, and users started to manipulate them and interact with them using the mouse.

This same process will repeat with Kinect. For now, interfaces are designed to be used with a mouse, a keyboard, a touchpad, or a touch screen. These interfaces are not ready for Kinect yet, and Kinect is not precise enough to move a cursor on a screen effectively—yet.

Computer interfaces must evolve, but before that happens, you can consider using Kinect to control the mouse—with some limitations.

First, it's probably not a good idea to try using Kinect in your office to control a mouse for work applications. The human body is not designed to make it easy for someone to keep both hands in the air for a long time, especially when sitting, so there are limits to what you can do to control an application you use sitting at a desk through the Kinect sensor.

However, you can think about using Kinect to control the mouse in situations where you can stand up, because in that position, it's easier to move your hands around.

In this chapter, you will discover how you can control the mouse pointer with Kinect, as well as how you can reduce the jitter (the undesired deviation of the values provided by the sensor) to get a smoother movement.

Controlling the mouse pointer

Before looking at the Kinect-specific code you need to control the mouse, you must import a Win32 function called *SendInput* (*http://pinvoke.net/default.aspx/user32/SendInput.html*). This function can send Windows messages to the topmost window on the computer screen to control inputs (mouse, keyboard, hardware). It uses some basic Win32 structures defined as follows in a file called enum.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;

namespace Kinect.Toolbox
{
    [Flags]
    internal enum MouseEventDataXButtons : uint
    {
        Nothing = 0x00000000,
        XBUTTON1 = 0x00000001,
        XBUTTON2 = 0x00000002
    }

    [Flags]
    internal enum MOUSEEVENTF : uint
    {
        ABSOLUTE = 0x8000,
        HWHEEL = 0x01000,
        MOVE = 0x0001,
        MOVE_NOCOALESCE = 0x2000,
        LEFTDOWN = 0x0002,
        LEFTUP = 0x0004,
        RIGHTDOWN = 0x0008,
        RIGHTUP = 0x0010,
        MIDDLEDOWN = 0x0020,
        MIDDLEUP = 0x0040,
        VIRTUALDESK = 0x4000,
        WHEEL = 0x0800,
        XDOWN = 0x0080,
        XUP = 0x0100
    }

    [StructLayout(LayoutKind.Sequential)]
    internal struct MOUSEINPUT
    {
        internal int dx;
        internal int dy;
        internal MouseEventDataXButtons mouseData;
        internal MOUSEEVENTF dwFlags;
        internal uint time;
        internal UIntPtr dwExtraInfo;
    }
}
```

The *MOUSEINPUT* structure provides two members (*dx* and *dy*) to define the mouse offset relative to its current position. It is included in a more global structure, *INPUT* (which you also have to include in your enum.cs file):

```
[StructLayout(LayoutKind.Explicit)]
internal struct INPUT
{
    [FieldOffset(0)]
    internal int type;
    [FieldOffset(4)]
    internal MOUSEINPUT mi;

    public static int Size
    {
        get { return Marshal.SizeOf(typeof(INPUT)); }
    }
}
```

The complete structure includes members for keyboard and hardware inputs, but they are not required for our application.

You can now create the following class to handle the control of the mouse:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;

namespace Kinect.Toolbox
{
       public static class MouseInterop
    {
        [DllImport("user32.dll")]
        private static extern uint SendInput(int nInputs, INPUT[] inputs, int size);

        static bool leftClickDown;

        public static void ControlMouse(int dx, int dy, bool leftClick)
        {
            INPUT[] inputs = new INPUT[2];

            inputs[0] = new INPUT();
            inputs[0].type = 0;
            inputs[0].mi.dx = dx;
            inputs[0].mi.dy = dy;
            inputs[0].mi.dwFlags = MOUSEEVENTF.MOVE;

            if (!leftClickDown && leftClick)
            {
                inputs[1] = new INPUT();
                inputs[1].type = 0;
                inputs[1].mi.dwFlags = MOUSEEVENTF.LEFTDOWN;
                leftClickDown = true;
            }
```

```
            else if (leftClickDown && !leftClick)
            {
                inputs[1] = new INPUT();
                inputs[1].type = 0;
                inputs[1].mi.dwFlags = MOUSEEVENTF.LEFTUP;
                leftClickDown = false;
            }

            SendInput(inputs.Length, inputs, INPUT.Size);
        }
    }
}
```

Notice that the code for the *ControlMouse* method is fairly obvious. You simply have to create a first *INPUT* value to set the mouse position.

Note Use *MOUSEEVENTF.MOVE* to specify position offsets and use *MOUSEEVENTF.ABSOLUTE* to specify absolute positions.

The second *INPUT* value is used to control the left mouse click. You use a *leftClickDown* boolean to handle the Up/Down events pair.

Using skeleton analysis to move the mouse pointer

Using the skeleton stream, you can track a hand and use its position to move the mouse. This is fairly simple to accomplish, because from the point of view of the screen, the position of the hand can be directly mapped to the screen space.

The basic approach

This first approach is fairly simple; you simply use the *Tools.Convert* method to get a two-dimensional (2D) projected value of the hand position:

```
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);
        }
```

Using this method, you can create the following class:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Windows.Forms; // You must add a reference to this assembly in your project
using System.Windows.Media.Media3D;
using Microsoft.Kinect;

namespace Kinect.Toolbox
```

```
{
    public class MouseController
    {
        Vector2? lastKnownPosition;

        public void SetHandPosition(KinectSensor sensor, Joint joint, Skeleton skeleton)
        {
            Vector2 vector2 = Tools.Convert(sensor, joint.Position);

            if (!lastKnownPosition.HasValue)
            {
                lastKnownPosition = vector2;
                return;
            }
         // GlobalSmooth will be added in the next part
            MouseInterop.ControlMouse((int)((vector2.X - lastKnownPosition.Value.X) * Screen.
PrimaryScreen.Bounds.Width * GlobalSmooth),
(int)((vector2.Y - lastKnownPosition.Value.Y) *
Screen.PrimaryScreen.Bounds.Height * GlobalSmooth), false);

            lastKnownPosition = vector2;
        }
    }
}
```

The main concern with this solution is the jitter produced by the Kinect sensor. It can lead to a really annoying experience because a user may have to work hard to maintain the proper position to control the mouse effectively.

Adding a smoothing filter

To improve the overall stability of the motion conducted by the Kinect sensor, you may decide you want to update your user interface. (You will learn more about doing this in Chapter 10, "Controls for Kinect.")

But you can also try to improve the smoothness of your data input by using a custom filter on the raw data. For instance, you can use the Holt Double Exponential Smoothing filter—there is a good description of the algorithm for this filter on Wikipedia at *http://en.wikipedia.org/wiki/Exponential_smoothing*.

The algorithm first tries to smooth the data by reducing the jitter and then by trying to find a trend in the past data to deduce new values.

Following is the code you need to add to the *MouseController* class in order to apply the filter:

```
// Filters
        Vector2 savedFilteredJointPosition;
        Vector2 savedTrend;
        Vector2 savedBasePosition;
        int frameCount;

        public float TrendSmoothingFactor
        {
```

```
    get;
    set;
}

public float JitterRadius
{
    get;
    set;
}

public float DataSmoothingFactor
{
    get;
    set;
}

public float PredictionFactor
{
    get;
    set;
}

public float GlobalSmooth
{
    get;
    set;
}

public MouseController()
{
    TrendSmoothingFactor = 0.25f;
    JitterRadius = 0.05f;
    DataSmoothingFactor = 0.5f;
    PredictionFactor = 0.5f;

    GlobalSmooth = 0.9f;
}

Vector2 FilterJointPosition(KinectSensor sensor, Joint joint)
{
    Vector2 filteredJointPosition;
    Vector2 differenceVector;
    Vector2 currentTrend;
    float distance;

    Vector2 baseJointPosition = Tools.Convert(sensor, joint.Position);
    Vector2 prevFilteredJointPosition = savedFilteredJointPosition;
    Vector2 previousTrend = savedTrend;
    Vector2 previousBaseJointPosition = savedBasePosition;

    // Checking frames count
    switch (frameCount)
    {
        case 0:
            filteredJointPosition = baseJointPosition;
            currentTrend = Vector2.Zero;
            break;
```

```
                case 1:
                    filteredJointPosition = (baseJointPosition + previousBaseJointPosition) *
0.5f;
                    differenceVector = filteredJointPosition - prevFilteredJointPosition;
                    currentTrend =
differenceVector * TrendSmoothingFactor + previousTrend * (1.0f - TrendSmoothingFactor);
                    break;
                default:
                    // Jitter filter
                    differenceVector = baseJointPosition - prevFilteredJointPosition;
                    distance = Math.Abs(differenceVector.Length());

                    if (distance <= JitterRadius)
                    {
                        filteredJointPosition = baseJointPosition * (distance / JitterRadius) +
prevFilteredJointPosition * (1.0f - (distance / JitterRadius));
                    }
                    else
                    {
                        filteredJointPosition = baseJointPosition;
                    }

                    // Double exponential smoothing filter
                    filteredJointPosition = filteredJointPosition *
(1.0f - DataSmoothingFactor) + (prevFilteredJointPosition + previousTrend) * DataSmoothingFactor;

                    differenceVector = filteredJointPosition - prevFilteredJointPosition;
                    currentTrend = differenceVector * TrendSmoothingFactor +
previousTrend * (1.0f - TrendSmoothingFactor);
                    break;
            }

            // Compute potential new position
            Vector2 potentialNewPosition = filteredJointPosition + currentTrend *
PredictionFactor;

            // Cache current value
            savedBasePosition = baseJointPosition;
            savedFilteredJointPosition = filteredJointPosition;
            savedTrend = currentTrend;
            frameCount++;

            return potentialNewPosition;
        }
```

This function can simply filter the joint position, so the *SetHandPosition* method will evolve to the following:

```
public void SetHandPosition(KinectSensor sensor, Joint joint, Skeleton skeleton)
{
    Vector2 vector2 = FilterJointPosition(sensor, joint);

    if (!lastKnownPosition.HasValue)
    {
        lastKnownPosition = vector2;
        return;
    }
```

```
    MouseInterop.ControlMouse((int)(
(vector2.X - lastKnownPosition.Value.X) * Screen.PrimaryScreen.Bounds.Width * GlobalSmooth),
(int)((vector2.Y - lastKnownPosition.Value.Y) * Screen.PrimaryScreen.Bounds.Height *
GlobalSmooth), false);

    lastKnownPosition = vector2;
}
```

Thanks to your brand-new filter code, the control of your mouse from the Kinect sensor should be less jittery, with fewer jerky movements. You will never achieve the precision of a handheld mouse, but it can work well with a large user interface.

You can use this filter in conjunction with the inbuilt *TransformSmoothParameters*. It can help smoothing the hand position in the three-dimensional (3D) space, and your filter can finish the work in the 2D space.

Handling the left mouse click

Creating a way to accomplish a common or left mouse click with input from the Kinect sensor is complex because if you decide to detect the click when the user pushes a hand (meaning the position of the hand on the z axis is changing), you can be almost certain that the user will also move the position of that hand on another axis at the same time, and so accordingly the mouse position will not be stable.

On the other hand, the implementation of this solution is simple:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Windows.Media.Media3D;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class MouseController
    {
        Vector2? lastKnownPosition;
        float previousDepth;

        public void SetHandPosition(KinectSensor sensor, Joint joint, Skeleton skeleton)
        {
            Vector2 vector2 = FilterJointPosition(sensor, joint);

            if (!lastKnownPosition.HasValue)
            {
                lastKnownPosition = vector2;
                previousDepth = joint.Position.Z;
                return;
            }
```

```
            var isClicked = Math.Abs(joint.Position.Z - previousDepth) > 0.05f;

            MouseInterop.ControlMouse((int)(
(vector2.X - lastKnownPosition.Value.X) * Screen.PrimaryScreen.Bounds.Width * GlobalSmooth),
(int)((vector2.Y - lastKnownPosition.Value.Y) * Screen.PrimaryScreen.Bounds.Height *
GlobalSmooth), isClicked);

            lastKnownPosition = vector2;
            previousDepth = joint.Position.Z;
        }
    }
}
```

You save the previous depth and compare it to the current value to determine if the difference is greater than a given threshold. But as mentioned previously, this is not the best solution. Chapter 10 provides you with a better way to handle mouse clicks.

If the interface has not been modified, you also can choose to use a gesture by another joint as a click trigger. In that case, you would update your class to this version:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Windows.Media.Media3D;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class MouseController
    {
        Vector2? lastKnownPosition;
        float previousDepth;

        // Filters
        Vector2 savedFilteredJointPosition;
        Vector2 savedTrend;
        Vector2 savedBasePosition;
        int frameCount;

        // Gesture detector for click
        GestureDetector clickGestureDetector;
        bool clickGestureDetected;
        public GestureDetector ClickGestureDetector
        {
            get
            {
                return clickGestureDetector;
            }
            set
            {
                if (value != null)
                {
                    value.OnGestureDetected += (obj) =>
```

```
                {
                    clickGestureDetected = true;
                };
        }

        clickGestureDetector = value;
    }
}

// Filter parameters
public float TrendSmoothingFactor
{
    get;
    set;
}

public float JitterRadius
{
    get;
    set;
}

public float DataSmoothingFactor
{
    get;
    set;
}

public float PredictionFactor
{
    get;
    set;
}

public float GlobalSmooth
{
    get;
    set;
}

public MouseController()
{
    TrendSmoothingFactor = 0.25f;
    JitterRadius = 0.05f;
    DataSmoothingFactor = 0.5f;
    PredictionFactor = 0.5f;

    GlobalSmooth = 0.9f;
}

Vector2 FilterJointPosition(KinectSensor sensor, Joint joint)
{
    Vector2 filteredJointPosition;
    Vector2 differenceVector;
    Vector2 currentTrend;
    float distance;
```

```
            Vector2 baseJointPosition = Tools.Convert(sensor, joint.Position);
            Vector2 prevFilteredJointPosition = savedFilteredJointPosition;
            Vector2 previousTrend = savedTrend;
            Vector2 previousBaseJointPosition = savedBasePosition;

            // Checking frames count
            switch (frameCount)
            {
                case 0:
                    filteredJointPosition = baseJointPosition;
                    currentTrend = Vector2.Zero;
                    break;
                case 1:
                    filteredJointPosition =
(baseJointPosition + previousBaseJointPosition) * 0.5f;
                    differenceVector = filteredJointPosition - prevFilteredJointPosition;
                    currentTrend =
differenceVector * TrendSmoothingFactor + previousTrend * (1.0f - TrendSmoothingFactor);
                    break;
                default:
                    // Jitter filter
                    differenceVector = baseJointPosition - prevFilteredJointPosition;
                    distance = Math.Abs(differenceVector.Length());

                    if (distance <= JitterRadius)
                    {
                        filteredJointPosition = baseJointPosition * (distance / JitterRadius) +
prevFilteredJointPosition * (1.0f - (distance / JitterRadius));
                    }
                    else
                    {
                        filteredJointPosition = baseJointPosition;
                    }

                    // Double exponential smoothing filter
                    filteredJointPosition = filteredJointPosition *
(1.0f - DataSmoothingFactor) + (prevFilteredJointPosition + previousTrend) * DataSmoothingFactor;

                    differenceVector = filteredJointPosition - prevFilteredJointPosition;
                    currentTrend =
differenceVector * TrendSmoothingFactor + previousTrend * (1.0f - TrendSmoothingFactor);
                    break;
            }

            // Compute potential new position
            Vector2 potentialNewPosition = filteredJointPosition + currentTrend *
PredictionFactor;

            // Cache current value
            savedBasePosition = baseJointPosition;
            savedFilteredJointPosition = filteredJointPosition;
            savedTrend = currentTrend;
            frameCount++;

            return potentialNewPosition;
        }

        public void SetHandPosition(KinectSensor sensor, Joint joint, Skeleton skeleton)
```

```
        {
            Vector2 vector2 = FilterJointPosition(sensor, joint);

            if (!lastKnownPosition.HasValue)
            {
                lastKnownPosition = vector2;
                previousDepth = joint.Position.Z;
                return;
            }

            bool isClicked;

            if (ClickGestureDetector == null)
                isClicked = Math.Abs(joint.Position.Z - previousDepth) > 0.05f;
            else
                isClicked = clickGestureDetected;

            MouseInterop.ControlMouse((int)(
(vector2.X - lastKnownPosition.Value.X) * Screen.PrimaryScreen.Bounds.Width * GlobalSmooth),
(int)((vector2.Y - lastKnownPosition.Value.Y) *
Screen.PrimaryScreen.Bounds.Height * GlobalSmooth), isClicked);

            lastKnownPosition = vector2;
            previousDepth = joint.Position.Z;

            clickGestureDetected = false;
        }
    }
}
```

This update tests the value of the *ClickGestureDetector* and then handles the *GestureDetected* event to virtually "detect" the click.

Ultimately, developers need to concentrate on building new user experiences that take advantage of all that Kinect has to offer, but if you want to add Kinect to an existing application, you now have the tools required to do so. Using these tools with the Kinect sensor, you are the mouse!

CHAPTER 10

Controls for Kinect

As you saw in Chapter 9, "You are the mouse!," it is important to adapt the user interface (UI) for your application to Kinect. You can use a Kinect sensor to control a standard mouse-oriented application, but it is far better to create an original Kinect-oriented application to take advantage of the unique characteristics of the sensor.

In this chapter, you will write a basic application designed to be controlled with a mouse or a Kinect sensor. As we clearly are in a transition period, it is important to support mouse-based UIs when possible, because this provides the ability to work around some of the difficulties of using Kinect (precision, user acceptance and understanding, etc.).

But if you want to create a complete Kinect user interface, do not hesitate to do so. Just be sure to take time to think about the usability and the discoverability of your user experience. Make sure that your user interface is as simple to use as a mouse. For now, users are accustomed to clicking a button or scrolling up and down using the mouse wheel. Because they are not currently aware of the conventions for using a Kinect sensor, you must make sure the controls are intuitive and easily understandable by users.

Your user interface must be natural. With a Kinect sensor, you remove the user's physical contact with the computer, so the experience must follow the natural habits of those users. For instance, if you want to control horizontal scrolling, it is most natural for the user to grab the current page and move it to the right or to the left with his hands to achieve that movement. Think about the most intuitive ways to accomplish an action with the user's body movements and gestures when you are designing the user interface.

Adapting the size of the elements

In the applications you create for Kinect, the best idea is to start by tracking the position of the hands of the user. Humans always use their hands to control the environment they are in or to support what they are saying, so it is natural for a user to choose to control the application with her hands.

This choice implies that the user wants to control a kind of mouse pointer with his or her dominant hand, for instance. This cursor will benefit from the same smoothing algorithm introduced in Chapter 9 to filter out the jitter—the Holt Double Exponential Smoothing filter. But you must also consider that even using the best smoothing algorithm, the precision of the movements of the cursor when

controlled by the Kinect sensor is not the same as when using a true mouse. So you have to create a user interface with larger controls, as shown in Figure 10-1.

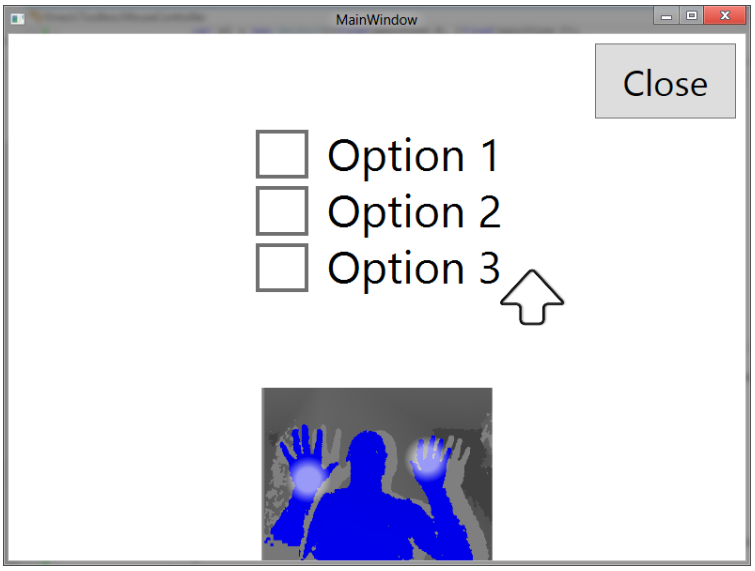

FIGURE 10-1 A sample user interface adapted for Kinect.

As you can see, the Kinect cursor (the arrow pointing up near Option 3) is bigger than your standard mouse pointer and the controls are larger to give the user more room to click them.

Providing specific feedback control

Even before you begin working on your user experience with Kinect, you must provide a visual feedback of the view of the user captured by the sensor.

This graphic feedback has been the subject of code created in previous chapters (such as Chapter 3, "Displaying Kinect data"), but for this application, you will write a reusable control based on the depth stream and the position of the user's hands. Using this control in your application will provide an immediate visual feedback for the user to view what the Kinect sensor is seeing and to determine if it is able to track the user's hands properly.

Figure 10-1 includes the visual feedback from this control as a frame inside the user interface for the sample application.

To create the visual feedback, first you must create a user control called *PresenceControl* with the following XAML:

```
<UserControl x:Class="Kinect.Toolbox.PresenceControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Grid.Resources>
            <RadialGradientBrush x:Key="radial" GradientOrigin="0.5, 0.5">
                <RadialGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0"/>
                    <GradientStop Color="White" Offset="0.5"/>
                    <GradientStop Color="Transparent" Offset="1.0"/>
                </RadialGradientBrush.GradientStops>
            </RadialGradientBrush>
        </Grid.Resources>
        <Image x:Name="image"/>
        <Ellipse x:Name="leftEllipse" Opacity="0.6" Width="50" Height="50"
VerticalAlignment="Top" HorizontalAlignment="Left" Margin="-25,-25,0,0"
Fill="{StaticResource radial}">
            <Ellipse.RenderTransform>
                <TranslateTransform x:Name="leftTransform"/>
            </Ellipse.RenderTransform>
        </Ellipse>
        <Ellipse x:Name="rightEllipse" Opacity="0.6" Width="50" Height="50"
VerticalAlignment="Top" HorizontalAlignment="Left" Margin="-25,-25,0,0"
Fill="{StaticResource radial}">
            <Ellipse.RenderTransform>
                <TranslateTransform x:Name="rightTransform"/>
            </Ellipse.RenderTransform>
        </Ellipse>
    </Grid>
</UserControl>
```

This control is based mainly on an image that is updated with the depth stream and two ellipses that move to indicate where the hands of the user should be positioned (you can see the ellipses as two light areas near the hands in the image shown at the bottom of Figure 10-1). The ellipses are filled with a radial gradient brush to simulate small glowing spots where the hands belong.

To update the image, you have to handle the *DepthFrameReady* event, and to update the position of the ellipses, you have to handle the *SkeletonFrameReady* event (as you did in the Chapter 3):

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

namespace Kinect.Toolbox
{
    public partial class PresenceControl : UserControl
    {
        KinectSensor kinectSensor;
        byte[] depthFrame32;
        short[] pixelData;
        WriteableBitmap bitmap;
```

```
        public PresenceControl()
        {
            InitializeComponent();
        }

        public void SetKinectSensor(KinectSensor sensor)
        {
            kinectSensor = sensor;

            kinectSensor.DepthFrameReady += kinectSensor_DepthFrameReady;
            kinectSensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);

            kinectSensor.SkeletonFrameReady += kinectSensor_SkeletonFrameReady;
        }
}
}
```

There is nothing new here—as mentioned, you have already used this kind of code.

The depth stream part of the code now should be obvious and even simpler than what was used in previous chapters, because you don't need to include the player index color:

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

                if (depthFrame32 == null)
                {
                    pixelData = new short[frame.PixelDataLength];
                    depthFrame32 = new byte[frame.Width * frame.Height * sizeof(int)];
                }

                frame.CopyPixelDataTo(pixelData);

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

                ConvertDepthFrame(pixelData);

                int stride = bitmap.PixelWidth * sizeof(int);
                Int32Rect dirtyRect =
new Int32Rect(0, 0, bitmap.PixelWidth, bitmap.PixelHeight);
                bitmap.WritePixels(dirtyRect, depthFrame32, stride, 0);
            }
        }

        void ConvertDepthFrame(short[] depthFrame16)
        {
            int i32 = 0;
```

```
            for (int i16 = 0; i16 < depthFrame16.Length; i16++)
            {
                int user = depthFrame16[i16] & 0x07;
                int realDepth = (depthFrame16[i16] >> DepthImageFrame.PlayerIndexBitmaskWidth);

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

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

                if (user > 0)
                {
                    depthFrame32[i32] = intensity;
                }
                else
                {
                    depthFrame32[i32] = (byte)(intensity / 2);
                    depthFrame32[i32 + 1] = (byte)(intensity / 2);
                    depthFrame32[i32 + 2] = (byte)(intensity / 2);
                }

                i32 += 4;
            }
        }
```

The skeleton part of the code will look for the position of the user's hands and will update the position of the ellipses in response, using the *MapSkeletonPointToDepth* method to compute screen coordinates:

```
void kinectSensor_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
{
    Skeleton[] skeletons = null;

    leftEllipse.Visibility = System.Windows.Visibility.Collapsed;
    rightEllipse.Visibility = System.Windows.Visibility.Collapsed;

    using (SkeletonFrame frame = e.OpenSkeletonFrame())
    {
        if (frame == null)
            return;

        Tools.GetSkeletons(frame, ref skeletons); // See Chapter 3, "Displaying Kinect data"

        if (skeletons.All(s => s.TrackingState == SkeletonTrackingState.NotTracked))
            return;
        foreach (var skeleton in skeletons)
        {
            if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
                continue;

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

```
                if (joint.JointType == JointType.HandRight)
                {
                    rightEllipse.Visibility = System.Windows.Visibility.Visible;
                    var handRightDepthPosition =
kinectSensor.MapSkeletonPointToDepth(joint.Position, DepthImageFormat.Resolution640x480Fps30);

                    rightTransform.X = (handRightDepthPosition.X / 640.0f) * Width;
                    rightTransform.Y = (handRightDepthPosition.Y / 480.0f) * Height;
                }
                else if (joint.JointType == JointType.HandLeft)
                {
                    leftEllipse.Visibility = System.Windows.Visibility.Visible;
                    var handLeftDepthPosition =
kinectSensor.MapSkeletonPointToDepth(joint.Position, DepthImageFormat.Resolution640x480Fps30);

                    leftTransform.X = (handLeftDepthPosition.X / 640.0f) * Width;
                    leftTransform.Y = (handLeftDepthPosition.Y / 480.0f) * Height;
                }
            }
        }
    }
}
```

With this control, you can now show your user whether or not he or she is currently being tracked by the Kinect sensor.

Replacing the mouse

The next step is to change the *MouseController* class you wrote in Chapter 9. You will draw a large custom pointer to replace it.

First you will create a user control called *MouseImpostor* to display the cursor. This control integrates a big arrow next to a hidden progress bar. (The progress bar will be used later to detect the click.) Indeed, instead of trying to simulate a click with a gesture (which is too complex for now), it is better to generate a click when the cursor is static during a few seconds (as in Xbox 360). To represent the time interval, the progress bar will fill to indicate to the user the remaining time before the click will be raised, as shown in Figure 10-2.

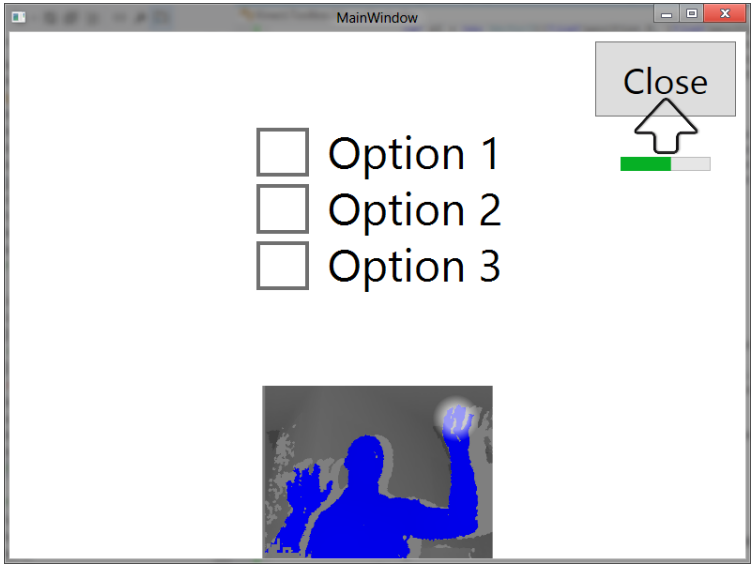

FIGURE 10-2 You can simulate a click by keeping the pointer on top of a control during a given duration.

The XAML of the user control is defined with the following code:

```
<UserControl x:Class="Kinect.Toolbox.MouseImpostor"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             Width="100"
             Height="100"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Image Source="pack://application:,,,/Kinect.Toolbox;component/Resources/Arrow.png"
Margin="10"/>
        <ProgressBar x:Name="progressBar" Height="15" VerticalAlignment="Bottom" Margin="2"
Minimum="0" Maximum="100" Visibility="Collapsed"/>
    </Grid>
</UserControl>
```

This control is simple: it consists of an image (which you must add to the assembly in a folder called Resources and define as *Embedded Resource*) and a progress bar.

Note If your project is not named *Kinect.Toolbox*, you will have to change the pack reference for the image. You can find a sample image in the companion content.

The code is also simple, with a *Progression* property used to fill and show (or hide) the progress bar:

```
using System;
using System.Windows.Controls;

namespace Kinect.Toolbox
{
    /// <summary>
    /// Interaction logic for MouseImpostor.xaml
    /// </summary>
    public partial class MouseImpostor : UserControl
    {
        public event Action OnProgressionCompleted;

        public MouseImpostor()
        {
            InitializeComponent();
        }

        public int Progression
        {
            set
            {
                if (value == 0 || value > 100)
                {
                    progressBar.Visibility = System.Windows.Visibility.Collapsed;
                    if (value > 100 && OnProgressionCompleted != null)
                        OnProgressionCompleted();
                }
                else
                {
                    progressBar.Visibility = System.Windows.Visibility.Visible;
                    progressBar.Value = value;

                }
            }
        }
    }
}
```

Note The *OnProgressionCompleted* event is raised when the progress bar is completely filled.

To use the new cursor you have created, you must update the *MouseController* class with the following new members:

```
// Impostors
using System;
using System.Windows.Media;
using System.Windows.Controls;

Canvas impostorCanvas;
```

```
Visual rootVisual;
MouseImpostor impostor;
```

The *impostorCanvas* is placed over all of the content on the client page to allow the cursor to display in the topmost layer. Thus, the code for a typical client page will be as follows:

```
<Window x:Class="KinectUI.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Kinect.Toolbox;assembly=Kinect.Toolbox"
        Loaded="MainWindow_Loaded_1"
        Height="600"
        Title="MainWindow" Width="800">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="184"/>
        </Grid.RowDefinitions>
        <StackPanel  HorizontalAlignment="Center" VerticalAlignment="Center">
            <CheckBox Content="Option 1">
                <CheckBox.LayoutTransform>
                    <ScaleTransform ScaleX="4" ScaleY="4"/>
                </CheckBox.LayoutTransform>
            </CheckBox>
            <CheckBox Content="Option 2">
                <CheckBox.LayoutTransform>
                    <ScaleTransform ScaleX="4" ScaleY="4"/>
                </CheckBox.LayoutTransform>
            </CheckBox>
            <CheckBox Content="Option 3">
                <CheckBox.LayoutTransform>
                    <ScaleTransform ScaleX="4" ScaleY="4"/>
                </CheckBox.LayoutTransform>
            </CheckBox>
        </StackPanel>
        <Button Content="Close" FontSize="38" HorizontalAlignment="Right" Width="150"
Height="80" VerticalAlignment="Top" Margin="10" Click="Button_Click_1"/>
        <local:PresenceControl Width="256" Height="184" VerticalAlignment="Bottom"
HorizontalAlignment="Center" x:Name="presenceControl" Grid.Row="1"/>
        <Canvas Grid.RowSpan="2" x:Name="mouseCanvas"/>
    </Grid>
</Window>
```

Notice that the *mouseCanvas* canvas overlaps all of the content of the root grid. The *rootVisual* visual is detected when the *impostorCanvas* is set. It contains the parent of the canvas (and the parent of all controls on the page).

After adding the previous members, you have to add the following property in the *MouseController* class to set all the required data:

```
public Canvas ImpostorCanvas
{
    set
    {
        if (value == null)
```

```
        {
            if (impostorCanvas != null)
                impostorCanvas.Children.Remove(impostor);

            impostor = null;
            rootVisual = null;
            return;
        }

        impostorCanvas = value;
        rootVisual = impostorCanvas.Parent as Visual;
        impostor = new MouseImpostor();

        value.Children.Add(impostor);
    }
}
```

To simplify the use of the *MouseController* class, you can add a simple singleton (a class that only allows one instance of itself to be created) to it:

```
static MouseController current;
public static MouseController Current
{
    get
    {
        if (current == null)
        {
            current = new MouseController();
        }

        return current;
    }
}
```

You create the connection between the client page and the *MouseController* class by adding the following code in the code behind the client page:

```
MouseController.Current.ImpostorCanvas = mouseCanvas;
```

Finally, you can update the *SetHandPosition* of the *MouseController* class to take in account the impostor:

```
public void SetHandPosition(KinectSensor sensor, Joint joint, Skeleton skeleton)
        {
            Vector2 vector2 = FilterJointPosition(sensor, joint);

            if (!lastKnownPosition.HasValue)
            {
                lastKnownPosition = vector2;
                previousDepth = joint.Position.Z;
                return;
            }

            var impostorPosition =
new Vector2((float)(vector2.X * impostorCanvas.ActualWidth), (float)(vector2.Y *
impostorCanvas.ActualHeight));
```

```
            bool isClicked = false;

                if (ClickGestureDetector == null)
                    isClicked = Math.Abs(joint.Position.Z - previousDepth) > 0.05f;
                else
                    isClicked = clickGestureDetected;


            if (impostor != null)
            {
                    Canvas.SetLeft(impostor, impostorPosition.X - impostor.ActualWidth / 2);
                    Canvas.SetTop(impostor, impostorPosition.Y);
            }
            else
            {
                MouseInterop.ControlMouse((int)(
(vector2.X - lastKnownPosition.Value.X) * Screen.PrimaryScreen.Bounds.Width * GlobalSmooth),
(int)((vector2.Y - lastKnownPosition.Value.Y) *
Screen.PrimaryScreen.Bounds.Height * GlobalSmooth), isClicked);
                lastKnownPosition = vector2;
            }

            previousDepth = joint.Position.Z;

            clickGestureDetected = false;
        }
```

According to the impostor member, the mouse pointer is updated directly or the *MouseController* class will use the cursor impostor control.

Magnetization!

The final concept you need to understand to complete your application is magnetization. Magnetization helps the user by automatically attracting the cursor when it is near a specific control, which helps offset the imprecision of the Kinect user interface.

The magnetized controls

First, you can add a simple collection of magnetized controls in the *MouseController*:

```
using System.Windows;

public List<FrameworkElement> MagneticsControl
{
    get;
    private set;
}
```

The magnetized range (the minimal distance required to attract the cursor) is also defined with a new property:

```
public float MagneticRange
{
    get;
    set;
}
```

Now you need to add some more class members:

```
bool isMagnetized;
DateTime magnetizationStartDate;
FrameworkElement previousMagnetizedElement;
```

Use the *isMagnetized* member to define whether a control is currently magnetizing the cursor. Use *magnetizationStartDate* to define when the cursor began to be static. The *previousMagnetizedElement* member defines the previous magnetized control. The constructor becomes:

```
MouseController()
{
    TrendSmoothingFactor = 0.25f;
    JitterRadius = 0.05f;
    DataSmoothingFactor = 0.5f;
    PredictionFactor = 0.5f;

    GlobalSmooth = 0.9f;

    MagneticsControl = new List<FrameworkElement>();

    MagneticRange = 25.0f;
}
```

The *SetHandPosition* has to evolve into a more complex method:

```
using System.Drawing;

public void SetHandPosition(KinectSensor sensor, Joint joint, Skeleton skeleton)
{
    Vector2 vector2 = FilterJointPosition(sensor, joint);

    if (!lastKnownPosition.HasValue)
    {
        lastKnownPosition = vector2;
        previousDepth = joint.Position.Z;
        return;
    }

    bool isClicked = false;

        if (ClickGestureDetector == null)
            isClicked = Math.Abs(joint.Position.Z - previousDepth) > 0.05f;
        else
            isClicked = clickGestureDetected;

    if (impostor != null)
    {
        // Still magnetized ?
```

```
        if ((vector2 - lastKnownPosition.Value).Length > 0.1f)
        {
            impostor.Progression = 0;
            isMagnetized = false;
            previousMagnetizedElement = null;
        }

        // Looking for nearest magnetic control
        float minDistance = float.MaxValue;
        FrameworkElement nearestElement = null;
        var impostorPosition =
new Vector2((float)(vector2.X * impostorCanvas.ActualWidth),
(float)(vector2.Y * impostorCanvas.ActualHeight));

        foreach (FrameworkElement element in MagneticsControl)
        {
            // Getting the four corners
            var position = element.TransformToAncestor(rootVisual).Transform(new Point(0, 0));
            var p1 = new Vector2((float)position.X, (float)position.Y);
            var p2 = new Vector2((float)(position.X + element.ActualWidth), (float)position.Y);
            var p3 = new Vector2((float)(position.X + element.ActualWidth), (float)(position.Y +
element.ActualHeight));
            var p4 = new Vector2((float)position.X, (float)(position.Y + element.ActualHeight));

            // Minimal distance
            float previousMinDistance = minDistance;
            minDistance = Math.Min(minDistance, (impostorPosition - p1).Length);
            minDistance = Math.Min(minDistance, (impostorPosition - p2).Length);
            minDistance = Math.Min(minDistance, (impostorPosition - p3).Length);
            minDistance = Math.Min(minDistance, (impostorPosition - p4).Length);

            if (minDistance != previousMinDistance)
            {
                nearestElement = element;
            }
        }

        // If a control is at a sufficient distance
        if (minDistance < MagneticRange || isMagnetized)
        {
            // Magnetic control found
            var position =
nearestElement.TransformToAncestor(rootVisual).Transform(new Point(0, 0));

            Canvas.SetLeft(impostor, position.X + nearestElement.ActualWidth / 2 -
impostor.ActualWidth / 2);
            Canvas.SetTop(impostor, position.Y + nearestElement.ActualHeight / 2);
            lastKnownPosition = vector2;

            if (!isMagnetized || previousMagnetizedElement != nearestElement)
            {
                isMagnetized = true;
                magnetizationStartDate = DateTime.Now;
            }
            else
            {
                impostor.Progression =
```

```
(int)(((DateTime.Now - magnetizationStartDate).TotalMilliseconds * 100) / 2000.0);
            }
        }
        else
        {
            Canvas.SetLeft(impostor, impostorPosition.X - impostor.ActualWidth / 2);
            Canvas.SetTop(impostor, impostorPosition.Y);
        }

        if (!isMagnetized)
            lastKnownPosition = vector2;

        previousMagnetizedElement = nearestElement;
    }
    else
    {
        MouseInterop.ControlMouse((int)((vector2.X - lastKnownPosition.Value.X) * Screen.
PrimaryScreen.Bounds.Width * GlobalSmooth), (int)((vector2.Y - lastKnownPosition.Value.Y) *
Screen.PrimaryScreen.Bounds.Height * GlobalSmooth), isClicked);
        lastKnownPosition = vector2;
    }
    previousDepth = joint.Position.Z;
    clickGestureDetected = false;
}
```

The intent here is to track the position of each corner and the relative distance of the corners to the cursor. If a control is near the cursor (meaning at a distance lesser than the *MagneticRange* property), the cursor is moved to the center of the control and the progression value of the impostor is updated, indicating that a click is in progress.

Simulating a click

You also need to change the *ImpostorCanvas* property to handle the *OnProgressionCompleted* of the impostor (to raise a click):

```
using System.Windows.Automation.Peers;
using System.Windows.Automation.Provider;

public Canvas ImpostorCanvas
{
    set
    {
        if (value == null)
        {
            if (impostorCanvas != null)
                impostorCanvas.Children.Remove(impostor);

            impostor.OnProgressionCompleted -= impostor_OnProgressionCompleted;

            impostor = null;
            rootVisual = null;
            return;
        }

        impostorCanvas = value;
        rootVisual = impostorCanvas.Parent as Visual;
```

```
        impostor = new MouseImpostor();

        impostor.OnProgressionCompleted += impostor_OnProgressionCompleted;

        value.Children.Add(impostor);
    }
}

void impostor_OnProgressionCompleted()
{
    if (previousMagnetizedElement != null)
    {
        var peer = UIElementAutomationPeer.CreatePeerForElement(previousMagnetizedElement);

        IInvokeProvider invokeProv =
peer.GetPattern(PatternInterface.Invoke) as IInvokeProvider;

        if (invokeProv == null)
        {
            var toggleProv = peer.GetPattern(PatternInterface.Toggle) as IToggleProvider;

            toggleProv.Toggle();
        }
        else
            invokeProv.Invoke();

        previousMagnetizedElement = null;
        isMagnetized = false;
    }
}
```

When the impostor raises the *OnProgressionCompleted* event, you can use the magic of the Automation assembly (reference the *UIAutomationProvider.dll* in your project). This namespace contains a class that helps you simulate the use of standard controls. For instance, the *IToggleProvider* interface allows you toggle a check box, and so on.

Adding a behavior to integrate easily with XAML

The last step you need to be able to use magnetization in the UI is to register controls as magnetizers. To do this, you can use an attached property to tag standard controls like this:

```
<CheckBox Content="Option 3" local:MagneticPropertyHolder.IsMagnetic="True">
    <CheckBox.LayoutTransform>
        <ScaleTransform ScaleX="4" ScaleY="4"/>
    </CheckBox.LayoutTransform>
</CheckBox>
```

The attached property is fairly simple, because it only adds the source control in the magnetizers list:

```
using System;
using System.Windows;

namespace Kinect.Toolbox
```

```
{
    public class MagneticPropertyHolder
    {
        public static readonly DependencyProperty IsMagneticProperty =
DependencyProperty.RegisterAttached("IsMagnetic", typeof(bool), typeof(MagneticPropertyHolder),
            new FrameworkPropertyMetadata(false, FrameworkPropertyMetadataOptions.AffectsRender,
OnIsMagneticChanged));

        static void OnIsMagneticChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
        {
            FrameworkElement element = d as FrameworkElement;
            if ((bool)e.NewValue)
            {
                if (!MouseController.Current.MagneticsControl.Contains(element))
                    MouseController.Current.MagneticsControl.Add(element);
            }
            else
            {
                if (MouseController.Current.MagneticsControl.Contains(element))
                    MouseController.Current.MagneticsControl.Remove(element);
            }
        }

        public static void SetIsMagnetic(UIElement element, Boolean value)
        {
            element.SetValue(IsMagneticProperty, value);
        }

        public static bool GetIsMagnetic(UIElement element)
        {
            return (bool)element.GetValue(IsMagneticProperty);
        }
    }
}
```

To make it as easy as possible to understand, here is the final version of the *MouseController* class:

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Automation.Peers;
using System.Windows.Automation.Provider;
using System.Windows.Controls;
using System.Windows.Forms;
using System.Windows.Media;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class MouseController
    {
        static MouseController current;
        public static MouseController Current
        {
            get
```

```
    {
        if (current == null)
        {
            current = new MouseController();
        }

        return current;
    }
}

Vector2? lastKnownPosition;
float previousDepth;

// Filters
Vector2 savedFilteredJointPosition;
Vector2 savedTrend;
Vector2 savedBasePosition;
int frameCount;

// Impostors
Canvas impostorCanvas;
Visual rootVisual;
MouseImpostor impostor;

bool isMagnetized;
DateTime magnetizationStartDate;
FrameworkElement previousMagnetizedElement;

// Gesture detector for click
GestureDetector clickGestureDetector;
bool clickGestureDetected;
public GestureDetector ClickGestureDetector
{
    get
    {
        return clickGestureDetector;
    }
    set
    {
        if (value != null)
        {
            value.OnGestureDetected += (obj) =>
                {
                    clickGestureDetected = true;
                };
        }

        clickGestureDetector = value;
    }
}

public Canvas ImpostorCanvas
{
    set
    {
        if (value == null)
        {
```

```
                    if (impostorCanvas != null)
                        impostorCanvas.Children.Remove(impostor);

                    impostor.OnProgressionCompleted -= impostor_OnProgressionCompleted;

                    impostor = null;
                    rootVisual = null;
                    return;
                }

                impostorCanvas = value;
                rootVisual = impostorCanvas.Parent as Visual;
                impostor = new MouseImpostor();

                impostor.OnProgressionCompleted += impostor_OnProgressionCompleted;

                value.Children.Add(impostor);
            }
        }

        void impostor_OnProgressionCompleted()
        {
            if (previousMagnetizedElement != null)
            {
                var peer =
UIElementAutomationPeer.CreatePeerForElement(previousMagnetizedElement);

                IInvokeProvider invokeProv =
peer.GetPattern(PatternInterface.Invoke) as IInvokeProvider;

                if (invokeProv == null)
                {
                    var toggleProv =
peer.GetPattern(PatternInterface.Toggle) as IToggleProvider;

                    toggleProv.Toggle();
                }
                else
                    invokeProv.Invoke();

                previousMagnetizedElement = null;
                isMagnetized = false;
            }
        }

        public float MagneticRange
        {
            get;
            set;
        }

        public List<FrameworkElement> MagneticsControl
        {
            get;
            private set;
        }
```

```
// Filter parameters
public float TrendSmoothingFactor
{
    get;
    set;
}

public float JitterRadius
{
    get;
    set;
}

public float DataSmoothingFactor
{
    get;
    set;
}

public float PredictionFactor
{
    get;
    set;
}

public float GlobalSmooth
{
    get;
    set;
}

MouseController()
{
    TrendSmoothingFactor = 0.25f;
    JitterRadius = 0.05f;
    DataSmoothingFactor = 0.5f;
    PredictionFactor = 0.5f;

    GlobalSmooth = 0.9f;

    MagneticsControl = new List<FrameworkElement>();

    MagneticRange = 25.0f;
}

Vector2 FilterJointPosition(KinectSensor sensor, Joint joint)
{
    Vector2 filteredJointPosition;
    Vector2 differenceVector;
    Vector2 currentTrend;
    float distance;

    Vector2 baseJointPosition = Tools.Convert(sensor, joint.Position);
    Vector2 prevFilteredJointPosition = savedFilteredJointPosition;
    Vector2 previousTrend = savedTrend;
    Vector2 previousBaseJointPosition = savedBasePosition;
```

```
            // Checking frames count
            switch (frameCount)
            {
                case 0:
                    filteredJointPosition = baseJointPosition;
                    currentTrend = Vector2.Zero;
                    break;
                case 1:
                    filteredJointPosition =
(baseJointPosition + previousBaseJointPosition) * 0.5f;
                    differenceVector = filteredJointPosition - prevFilteredJointPosition;
                    currentTrend = differenceVector * TrendSmoothingFactor + previousTrend *
(1.0f - TrendSmoothingFactor);
                    break;
                default:
                    // Jitter filter
                    differenceVector = baseJointPosition - prevFilteredJointPosition;
                    distance = Math.Abs(differenceVector.Length);

                    if (distance <= JitterRadius)
                    {
                        filteredJointPosition = baseJointPosition * (distance / JitterRadius) +
prevFilteredJointPosition * (1.0f - (distance / JitterRadius));
                    }
                    else
                    {
                        filteredJointPosition = baseJointPosition;
                    }

                    // Double exponential smoothing filter
                    filteredJointPosition = filteredJointPosition * (1.0f - DataSmoothingFactor)
+ (prevFilteredJointPosition + previousTrend) * DataSmoothingFactor;

                    differenceVector = filteredJointPosition - prevFilteredJointPosition;
                    currentTrend = differenceVector * TrendSmoothingFactor + previousTrend *
(1.0f - TrendSmoothingFactor);
                    break;
            }

            // Compute potential new position
            Vector2 potentialNewPosition = filteredJointPosition + currentTrend *
PredictionFactor;

            // Cache current value
            savedBasePosition = baseJointPosition;
            savedFilteredJointPosition = filteredJointPosition;
            savedTrend = currentTrend;
            frameCount++;

            return potentialNewPosition;
        }

        public void SetHandPosition(KinectSensor sensor, Joint joint, Skeleton skeleton)
        {
            Vector2 vector2 = FilterJointPosition(sensor, joint);

            if (!lastKnownPosition.HasValue)
```

```
            {
                lastKnownPosition = vector2;
                previousDepth = joint.Position.Z;
                return;
            }

            bool isClicked = false;


            if (ClickGestureDetector == null)
                isClicked = Math.Abs(joint.Position.Z - previousDepth) > 0.05f;
            else
               isClicked = clickGestureDetected;

            if (impostor != null)
            {
                // Still magnetized ?
                if ((vector2 - lastKnownPosition.Value).Length > 0.1f)
                {
                    impostor.Progression = 0;
                    isMagnetized = false;
                    previousMagnetizedElement = null;
                }

                // Looking for nearest magnetic control
                float minDistance = float.MaxValue;
                FrameworkElement nearestElement = null;
                var impostorPosition = new Vector2(
(float)(vector2.X * impostorCanvas.ActualWidth),
(float)(vector2.Y * impostorCanvas.ActualHeight));

                foreach (FrameworkElement element in MagneticsControl)
                {
                    // Getting the four corners
                    var position =
element.TransformToAncestor(rootVisual).Transform(new Point(0, 0));
                    var p1 = new Vector2((float)position.X, (float)position.Y);
                    var p2 = new Vector2((float)(position.X + element.ActualWidth),
(float)position.Y);
                    var p3 = new Vector2((float)(position.X + element.ActualWidth),
(float)(position.Y + element.ActualHeight));
                    var p4 = new Vector2((float)position.X,
(float)(position.Y + element.ActualHeight));

                    // Minimal distance
                    float previousMinDistance = minDistance;
                    minDistance = Math.Min(minDistance, (impostorPosition - p1).Length);
                    minDistance = Math.Min(minDistance, (impostorPosition - p2).Length);
                    minDistance = Math.Min(minDistance, (impostorPosition - p3).Length);
                    minDistance = Math.Min(minDistance, (impostorPosition - p4).Length);

                    if (minDistance != previousMinDistance)
                    {
                        nearestElement = element;
                    }
                }
```

```
                // If a control is at a sufficient distance
                if (minDistance < MagneticRange || isMagnetized)
                {
                    // Magnetic control found
                    var position =
nearestElement.TransformToAncestor(rootVisual).Transform(new Point(0, 0));

                    Canvas.SetLeft(impostor, position.X +
nearestElement.ActualWidth / 2 - impostor.ActualWidth / 2);
                    Canvas.SetTop(impostor, position.Y + nearestElement.ActualHeight / 2);
                    lastKnownPosition = vector2;

                    if (!isMagnetized || previousMagnetizedElement != nearestElement)
                    {
                        isMagnetized = true;
                        magnetizationStartDate = DateTime.Now;
                    }
                    else
                    {
                        impostor.Progression =
(int)(((DateTime.Now - magnetizationStartDate).TotalMilliseconds * 100) / 2000.0);
                    }
                }
                else
                {
                    Canvas.SetLeft(impostor, impostorPosition.X - impostor.ActualWidth / 2);
                    Canvas.SetTop(impostor, impostorPosition.Y);
                }

                if (!isMagnetized)
                    lastKnownPosition = vector2;

                previousMagnetizedElement = nearestElement;
            }
            else
            {
                MouseInterop.ControlMouse((int)
((vector2.X - lastKnownPosition.Value.X) * Screen.PrimaryScreen.Bounds.Width * GlobalSmooth),
(int)((vector2.Y - lastKnownPosition.Value.Y) *
Screen.PrimaryScreen.Bounds.Height * GlobalSmooth), isClicked);
                lastKnownPosition = vector2;
            }


            previousDepth = joint.Position.Z;

            clickGestureDetected = false;
        }
    }
}
```

You now have everything you need to create a Kinect-ready application. Using these new controls and tools, you can introduce a more adapted user interface that takes into account the precision available when working with the Kinect sensor.

CHAPTER 11

Creating augmented reality with Kinect

This book would not be complete without a chapter about augmented reality. The term *augmented reality* defines a process based on a real-time video with additional information added on top of the video (information such as the name of the person in the video, the position of a building or landmark, a 3D object, and so forth).

To be able to create applications that take advantage of augmented reality, you must find a way to project every pixel to a three-dimensional (3D) coordinates system to locate the pixels geographically and add relevant information. And as you learned in previous chapters, doing this conversion with Kinect is straightforward and simple!

In this chapter, you will use the Kinect sensor to create a fun augmented reality application that adds a lightsaber as an extension of your hand, as shown in Figure 11-1.

FIGURE 11-1 Adding a lightsaber on top of the image in a video.

More Info This chapter uses XNA and 3D features intensively. Feel free to go to *http://msdn.microsoft.com/en-us/library/bb200104.aspx* to learn more about these topics, which are beyond the scope of this book.

Creating the XNA project

To provide the necessary support to create a real-time 3D image, it makes sense to use an XNA project as the basis for the lightsaber application. XNA is a .NET assembly that forms a very powerful wrapper on top of DirectX.

Using it, you can create a real-time 3D rendering application that works with Windows, Xbox 360, and Windows Phone operating systems.

More Info Download XNA at *http://www.microsoft.com/en-us/download/details.aspx?id=23714*.

After installing XNA, you can create a simple Windows Game application with it. Figure 11-2 shows the Add New Project feature you use to begin creating a new application.

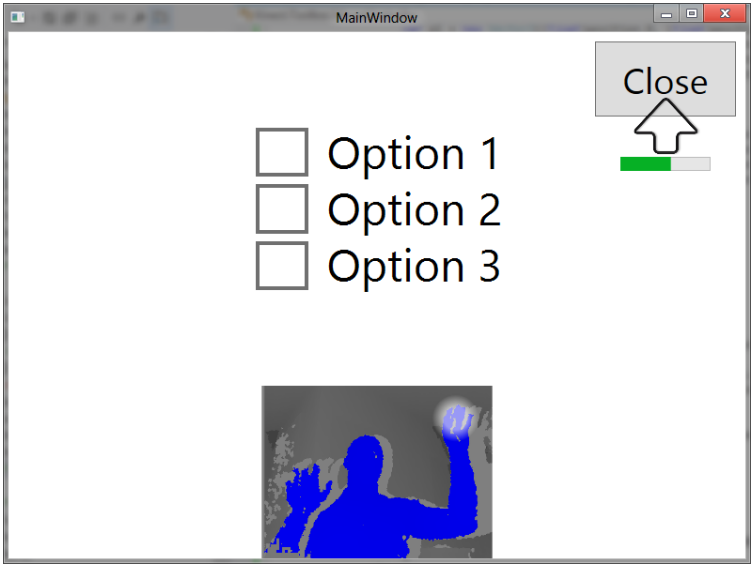

FIGURE 11-2 Creating an XNA project for Windows.

The application contains two projects—one for the game itself, and one for the content (pictures, sound, etc.), as shown in Figure 11-3.

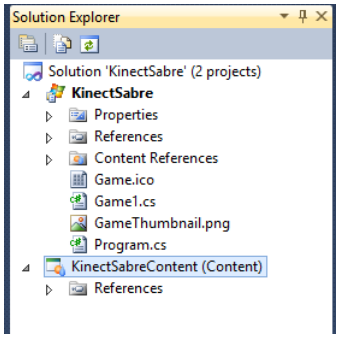

FIGURE 11-3 A basic XNA solution with a game project (*KinectSabre*) and a content project (*KinectSabreContent*).

For now, all the code is inside the *Game1.cs* class and can be simplified as follows:

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace KinectSabre
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        readonly GraphicsDeviceManager graphics;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this)
            {
                PreferredBackBufferWidth = 800,
                PreferredBackBufferHeight = 600,
                GraphicsProfile = GraphicsProfile.HiDef,
            };

            IsFixedTimeStep = true;
            TargetElapsedTime = TimeSpan.FromMilliseconds(30);

            Content.RootDirectory = "Content";
        }

        protected override void Update(GameTime gameTime)
        {
            if (GraphicsDevice == null)
                return;

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.Black);

            // Base
            base.Draw(gameTime);
        }
    }
}
```

The main part of the code is inside the *Game1* constructor, where it initializes an 800 × 600 display with a refresh rate of 30 frames per second. The *HiDef* profile will be useful later when you will have to deal with shaders (for special effects). You need a current graphics card to handle the *HiDef* profile correctly. If you do not have an up-to-date graphics card, you can use the *LoDef* profile, but in that case the glow around the lightsaber will not work.

XNA works mainly with two methods: *Draw* and *Update*.

- *Draw* is called when the game determines it is time to draw a frame.
- *Update* is called when the game determines that game logic needs to be processed.

It is your responsibility to add code in these two methods. For now, the application simply displays a black screen, but it is a really fast black screen because XNA uses all the power of your graphics card!

Connecting to a Kinect sensor

Now that you have created the foundation of your application, you have to connect to a Kinect sensor. The code required to do this is strictly the same as the basic code you created in Chapter 1, "A bit of background":

```
KinectSensor kinectSensor;
 void LaunchKinect()
 {
     try
     {
         kinectSensor = KinectSensor.KinectSensors[0];

         kinectSensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
         kinectSensor.SkeletonStream.Enable(new TransformSmoothParameters()
         {
             Smoothing = 0.5f,
             Correction = 0.5f,
             Prediction = 0.5f,
             JitterRadius = 0.05f,
             MaxDeviationRadius = 0.04f
         });

         kinectSensor.Start();

         Window.Title = "KinectSabre - Sensor connected";
     }
     catch
     {
         kinectSensor = null;
         Exit();
     }
 }
```

This code creates a link with the Kinect sensor and activates the color and skeleton streams. But it does not add event handlers to get frames. You will use a pulling mechanism to grab these frames when they are required.

Adding the background

The next step in creating the lightsaber involves adding a video background to your application. To do so, you need *SpriteBatch* and *Texture2D* objects. *SpriteBatch* is an XNA object used to display sprites and *Texture2D* is an XNA object used to store pictures.

Add the following code to your *Game1.cs* class:

```
SpriteBatch spriteBatch;
Texture2D colorTexture;
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    colorTexture = new Texture2D(GraphicsDevice, 640, 480, false, SurfaceFormat.Color);
}
```

The *LoadContent* method is automatically called at startup by XNA. The *colorTexture* variable will be used to display the Kinect video.

You then must update the *Draw* method to use the *spriteBatch* to draw the *colorTexture* variable:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    GraphicsDevice.BlendState = BlendState.Opaque;
    spriteBatch.Begin();
    spriteBatch.Draw(colorTexture, new Rectangle(0, 0, GraphicsDevice.Viewport.Width,
GraphicsDevice.Viewport.Height), Color.White);
    spriteBatch.End();
    // Base
    base.Draw(gameTime);
}
```

To copy the content of a Kinect video frame to your *colorTexture* variable, you must create a *GetVideoFrame* method like this one:

```
byte[] bits;
void GetVideoFrame()
{
    if (kinectSensor == null)
        return;

    var frame = kinectSensor.ColorStream.OpenNextFrame(0);

    if (frame == null)
        return;
```

```
    if (bits == null || bits.Length != frame.PixelDataLength)
    {
        bits = new byte[frame.PixelDataLength];
    }
    frame.CopyPixelDataTo(bits);

    UpdateColorTexture(bits);

    // Reset textures used to null in order to update it
    GraphicsDevice.Textures[0] = null;
    GraphicsDevice.Textures[1] = null;
    GraphicsDevice.Textures[2] = null;

    // Update the texture
    colorTexture.SetData(colorBuffer);
}

private byte[] colorBuffer;
public void UpdateColorTexture(byte[] bits)
{
    if (colorBuffer == null)
        colorBuffer = new byte[640 * 480 * 4];

    for (int index = 0; index < colorTexture.Width * colorTexture.Height; index++)
    {
        byte b = bits[index * 4];
        byte g = bits[index * 4 + 1];
        byte r = bits[index * 4 + 2];

        colorBuffer[index * 4] = r;
        colorBuffer[index * 4 + 1] = g;
        colorBuffer[index * 4 + 2] = b;
        colorBuffer[index * 4 + 3] = 0;
    }
}
```

So the *GetVideoFrame* method queries the current Kinect video frame and retrieves all the associated bits. Using this information, it calls the *UpdateColorTexture* method, which is responsible for creating an RGBA buffer named *colorBuffer* from the original BGR data. Finally, the *colorBuffer* buffer is transmitted to the *colorTexture* variable.

Next you have to change your *Update* method to call the *GetVideoFrame*:

```
protected override void Update(GameTime gameTime)
{
    if (GraphicsDevice == null)
        return;
    GetVideoFrame();

    base.Update(gameTime);
}
```

Figure 11-4 shows you the result of using the application.

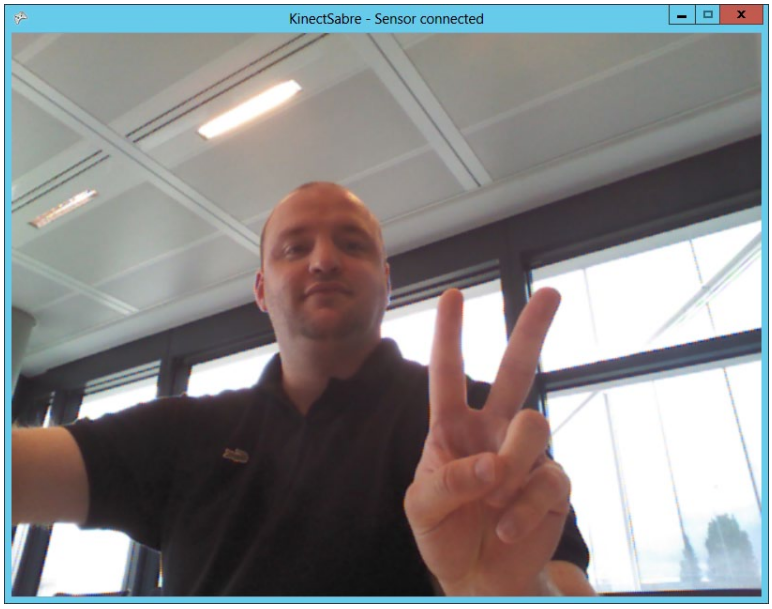

FIGURE 11-4 Display the Kinect video stream with *SpriteBatch* and *Texture2D*.

Adding the lightsaber

The next step in creating your augmented reality application involves adding the lightsaber on top of the video background acquired from the Kinect sensor.

Creating the saber shape

To add the shape of the saber, you can use a stretched 3D box, which is a simple object to create. First you have to create a new class named *VertexPositionColor* to store the definition of each point of the cube:

```
using System;
using System.Runtime.InteropServices;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace KinectSabre
{
    [Serializable]
    [StructLayout(LayoutKind.Sequential)]
    public struct VertexPositionColor : IVertexType
    {
        public Vector3 Position;
        public Vector4 Color;

        public static readonly VertexDeclaration VertexDeclaration;
```

```
        public VertexPositionColor(Vector3 pos, Vector4 color)
        {
            Position = pos;
            Color = color;
        }

        VertexDeclaration IVertexType.VertexDeclaration
        {
            get { return VertexDeclaration; }
        }

        static VertexPositionColor()
        {
            VertexElement[] elements = new VertexElement[]
            {
                new VertexElement(0, VertexElementFormat.Vector3,
VertexElementUsage.Position,0),
                new VertexElement(12, VertexElementFormat.Vector4, VertexElementUsage.Color,0)
            };
            VertexDeclaration = new VertexDeclaration(elements);
        }
    }
}
```

This structure defines that a vertex (a 3D point) is composed of a position and a color. It also contains a *VertexDeclaration* object to describe the structure to XNA.

Using this structure, you can add a new class called *Cube.cs*:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace KinectSabre
{
    public class Cube
    {
        BasicEffect basicEffect;
        GraphicsDevice device;

        public Cube(GraphicsDevice device)
        {
            this.device = device;
            World = Matrix.Identity;
            basicEffect = new BasicEffect(device);

            CreateBuffers();
        }

        private static readonly VertexPositionColor[] _vertices = new[]
        {
            //right
            new VertexPositionColor(new Vector3( 1,  1,  1), Color.White.ToVector4()),
            new VertexPositionColor(new Vector3( 1,  1, -1), Color.White.ToVector4()),
            new VertexPositionColor(new Vector3( 1, 0,  1), Color.White.ToVector4()),
            new VertexPositionColor(new Vector3( 1, 0, -1), Color.White.ToVector4()),
```

```
            //left
            new VertexPositionColor(new Vector3(-1,  1, -1), Color.White.ToVector4()),
            new VertexPositionColor(new Vector3(-1,  1,  1), Color.White.ToVector4()),
            new VertexPositionColor(new Vector3(-1, 0, -1), Color.White.ToVector4()),
            new VertexPositionColor(new Vector3(-1, 0,  1), Color.White.ToVector4())
        };

        static readonly int[] _indices = new[]
        {
            0,1,2,
            2,1,3,

            4,5,6,
            6,5,7,

            5,1,0,
            5,4,1,

            7,3,2,
            7,6,3,

            4,3,1,
            4,6,3,

            5,0,2,
            5,7,2
        };

        private VertexBuffer _vb;
        private IndexBuffer _ib;

        public void CreateBuffers()
        {
            _vb = new VertexBuffer(device, VertexPositionColor.VertexDeclaration,
_vertices.Length, BufferUsage.None);
            _vb.SetData(_vertices);
            _ib = new IndexBuffer(device, IndexElementSize.ThirtyTwoBits,
_indices.Length, BufferUsage.None);
            _ib.SetData(_indices);
        }

        public Matrix World { get; set; }
        public Matrix View { get; set; }
        public Matrix Projection { get; set; }

        public void Draw()
        {
            basicEffect.World = World;
            basicEffect.View = View;
            basicEffect.Projection = Projection;
            basicEffect.VertexColorEnabled = true;

            device.SetVertexBuffer(_vb);
            device.Indices = _ib;

            foreach (var pass in basicEffect.CurrentTechnique.Passes)
            {
```

```
                pass.Apply();
                device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
_vb.VertexCount, 0, _indices.Length / 3);
            }
        }
    }
}
```

This class is mainly XNA code, which is out of the scope of this book, but it can be summarized as follows:

- It creates a list of eight white vertices positioned in the eight corners.
- It creates a list of faces (composed by three indices in the vertices list).
- It creates a vertex buffer and an index buffer (the list of the points and the list of the faces).
- It creates a *basicEffect* to draw the cube.
- It creates a *Draw* method.
- It allows the user to specify a *World* matrix (for the object), a *View* matrix (for the camera), and a *Projection* matrix (for the projection to the screen).

More Info You can find more information about the class XNA code at *http://msdn.microsoft.com/en-us/library/bb203933(v=xnagamestudio.40)*.

Using this class is a fairly straightforward step in creating the application. You simply instantiate it in the *Game1.LoadContent* method, and then you use the new object in your *Game1.Draw* method:

```
Cube sabre;
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    colorTexture = new Texture2D(GraphicsDevice, 640, 480, false, SurfaceFormat.Color);

    sabre = new Cube(GraphicsDevice);
}

void DrawSabre(Vector3 point, Matrix orientationMatrix)
{
    sabre.View = Matrix.Identity;
    sabre.Projection = Matrix.CreatePerspectiveFieldOfView(0.87f,
GraphicsDevice.Viewport.AspectRatio, 0.1f, 1000.0f);

    sabre.World = Matrix.CreateScale(0.01f, 1.3f, 0.01f) * orientationMatrix * Matrix.
CreateTranslation(point * new Vector3(1, 1, -1f));
    sabre.Draw();
}

protected override void Draw(GameTime gameTime)
{
```

```
    GraphicsDevice.Clear(Color.Black);

    GraphicsDevice.BlendState = BlendState.Opaque;
    spriteBatch.Begin();
    spriteBatch.Draw(colorTexture, new Rectangle(0, 0, GraphicsDevice.Viewport.Width,
GraphicsDevice.Viewport.Height), Color.White);
    spriteBatch.End();

    DrawSabre(new Vector3(0, 0, 0.5f), Matrix.Identity);

    // Base
    base.Draw(gameTime);
}
```

The main part of this code to note here is the *DrawSabre* method, which stretches the *Cube* object with a scale matrix. It also takes a position and a rotation as parameters. You will use these parameters later to synchronize the position and the orientation of the saber with the position and the orientation of your hand.

Figure 11-5 shows the image that results when you add the basic saber object.

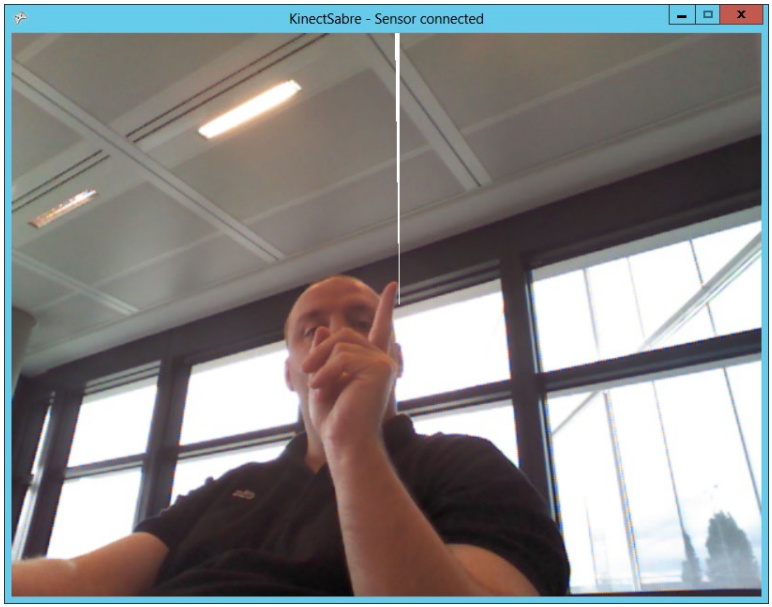

FIGURE 11-5 The first display of the lightsaber: a stretched cube positioned at the center of the screen.

Controlling the saber

To control the position and the orientation of the lightsaber, you use the Kinect skeleton stream to look for the position of the hand that will hold the saber. To make this application even more fun, you can support up to two players by handling two tracked skeletons—that is, you can effectively create a lightsaber duel.

So, the main job of the following code (which you need to add in the *Game1* class) is to detect and store the position and orientations of the saber-holding hand of each player:

```
using System.Linq;
Skeleton[] skeletons;

Vector3 p1LeftHandPosition { get; set; }
Matrix p1LeftHandMatrix { get; set; }
bool p1IsActive { get; set; }

Vector3 p2LeftHandPosition { get; set; }
Matrix p2LeftHandMatrix { get; set; }
bool p2IsActive { get; set; }

void GetSkeletonFrame()
{
    if (kinectSensor == null)
        return;

    SkeletonFrame frame = kinectSensor.SkeletonStream.OpenNextFrame(0);

    if (frame == null)
        return;

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

    bool player1 = true;

    foreach (Skeleton data in skeletons)
    {
        if (data.TrackingState == SkeletonTrackingState.Tracked)
        {
            foreach (Joint joint in data.Joints)
            {
                // Quality check
                if (joint.TrackingState != JointTrackingState.Tracked)
                    continue;

                switch (joint.JointType)
                {
                    case JointType.HandLeft:
                        if (player1)
                        {
                            p1LeftHandPosition = joint.Position.ToVector3();
                            p1LeftHandMatrix = data.BoneOrientations.Where(
b => b.EndJoint == JointType.HandLeft)
.Select(b => b.AbsoluteRotation).FirstOrDefault().Matrix.ToMatrix();
                        }
                        else
                        {
                            p2LeftHandPosition = joint.Position.ToVector3();
                            p2LeftHandMatrix = data.BoneOrientations.Where(
```

```
b => b.EndJoint == JointType.HandLeft)
.Select(b => b.AbsoluteRotation).FirstOrDefault().Matrix.ToMatrix();
                        }
                        break;
                    }
                }

                if (player1)
                {
                    player1 = false;
                    p1IsActive = true;
                }
                else
                {
                    p2IsActive = true;
                    return;
                }
            }
        }

        if (player1)
            p1IsActive = false;

        p2IsActive = false;
    }
```

To convert from Kinect space to XNA 3D space, this code uses some extension methods, defined as follows:

```
using Microsoft.Kinect;
using Microsoft.Xna.Framework;

namespace KinectSabre
{
    public static class Tools
    {
        public static Vector3 ToVector3(this SkeletonPoint vector)
        {
            return new Vector3(vector.X, vector.Y, vector.Z);
        }

        public static Matrix ToMatrix(this Matrix4 value)
        {
            return new Matrix(  value.M11, value.M12, value.M13, value.M14,
                                value.M21, value.M22, value.M23, value.M24,
                                value.M31, value.M32, value.M33, value.M34,
                                value.M41, value.M42, value.M43, value.M44);
        }
    }
}
```

Note that the orientation matrix and the position vector are already expressed in the real-world coordinates (relative to the sensor), so you do not need to provide code to compute the saber's position and orientation (a direct mapping takes care of this!).

To take this new code into account, simply update the *Draw* method as follows:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    GraphicsDevice.BlendState = BlendState.Opaque;
    spriteBatch.Begin();
    spriteBatch.Draw(colorTexture, new Rectangle(0, 0, GraphicsDevice.Viewport.Width,
GraphicsDevice.Viewport.Height), Color.White);
    spriteBatch.End();

    // Sabre
    GetSkeletonFrame();

    if (p1IsActive)
        DrawSabre(p1LeftHandPosition, p1LeftHandMatrix);

    if (p2IsActive)
        DrawSabre(p2LeftHandPosition, p2LeftHandMatrix);

    // Base
    base.Draw(gameTime);
}
```

The resulting application is starting to look good, as shown in Figure 11-6.

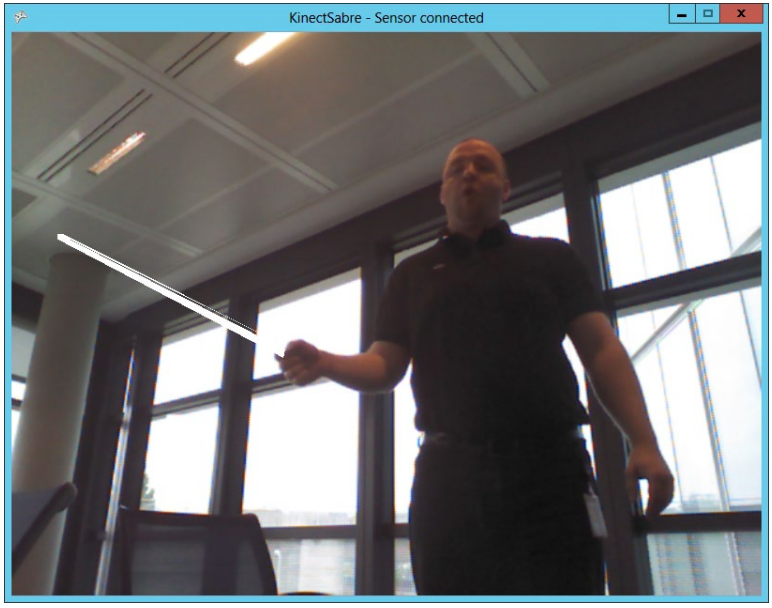

FIGURE 11-6 Basic integration of the lightsaber.

Creating a "lightsaber" effect

The final step in creating the lightsaber application involves adding a red glow effect around the saber. To do this, you must apply a shader effect to the image produced by XNA.

Shaders are programs for the graphics card that allow the developer to manipulate the pixels (pixel shader) and the vertices (vertex shader). They are developed using a language called HLSL (High Level Shader Language), which is a language similar to C.

Adding the glow effect gives you the result shown in Figure 11-7. (The glow appears as a medium gray halo around the saber shape in the print book.)

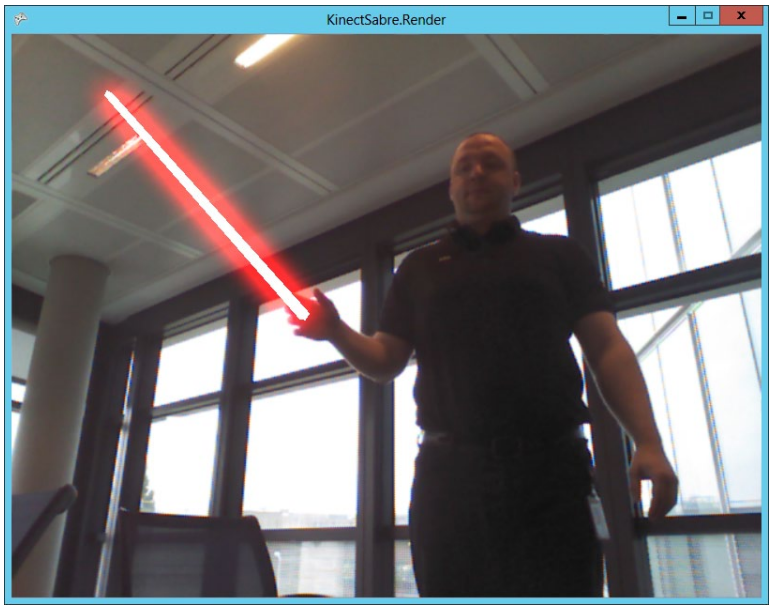

FIGURE 11-7 The final effect showing the red glow around the 3D box to create the lightsaber.

Once again, shaders are not really within the scope of this book, but it is an important effect that will make your lightsaber application look authentic. So, to make your lightsaber light up, you can download the previous code associated with the glow effect at *http://www.catuhe.com/book/lightsaber.zip*.

Going further

Throughout these chapters, you have learned a lot about Kinect for Windows SDK. You learned how to use the SDK to detect gestures and postures. You learned how to provide adapted user interface for Kinect sensors. You also learned how to use combination of video and 3D to create augmented reality applications.

Now, I suggest you go to *http://kinecttoolbox.codeplex.com* to follow the evolution of the library you created during this book. Feel free to use it in your own application to create the user interface of tomorrow!

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

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.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press