Hilo: Developing C++ Applications for Windows 7

Introduction

"Hilo" is a series of articles and sample applications that show how you can leverage the power of Windows 7, using the Visual Studio 2010 and Visual C++ development systems to build high performance, responsive rich client applications. Hilo provides both source code and written guidance to help you design and develop Windows applications of your own.

The series covers many topics, including the key capabilities and features of Windows 7, the design process for the user experience, and application design and architecture. Source code is provided so that you can see firsthand how the accompanying sample applications were designed and implemented. You can also use the source code in your own projects to produce your own rich, compelling applications for Windows 7. The Hilo sample applications are designed for high performance and responsiveness and are written entirely in C++ using Visual C++.

These articles describe the design and implementation of a set of touch-enabled applications that allow you to browse, select, and work with images. They will illustrate how to write applications that leverage some of the powerful capabilities that Windows 7 provides. You will see how the various technologies for Windows 7 can be used together to create a compelling user experience.

Chapter	Description
1: Introducing Hilo	The first Hilo sample application—the Hilo Browser—implements a touch-enabled user interface for browsing and selecting photos and images.
2: Setting up the Hilo Development Environment	This chapter outlines how to set up a workstation for the development environment so that you can compile and run the Hilo Browser sample application.
3: Choosing Windows Development Technologies	This chapter describes the rationale for the choice of the development technologies used to implement the Hilo applications.
4: Designing the Hilo User Experience	This chapter describes the process and thoughts when developing the Hilo User Experience.
5: The Hilo Common Library	This chapter introduces the Hilo Common Library, a lightweight object orientated library to help to create and manage Hilo-based application windows and handle messages sent to them.
6: Using Windows Direct2D	This chapter describes how hardware accelerated Direct2D and DirectWrite are used in the Hilo sample application.
7: Using Windows Animation Manager	This chapter explores the Windows 7 Windows Animation Manager, that handles the complexities of image changes over time.
8: Using Windows 7 Libraries and	Files from many different locations can be accessed through a single logical location according to their type even though they are stored in many different locations.

the Shell	Libraries are user defined collections of content that are indexed to enable faster search and sorting. Hilo uses the Windows 7 Libraries feature to access the user's images.
9: Introducing Hilo Annotator	This chapter describes the Hilo Annotator application, which allows you to crop, rotate, and draw on the photographs you have selected. Hilo Annotator uses the Windows Ribbon Control to provide easy access to the various annotation functions, and the Windows Imaging Component to load and manipulate the images and their metadata.
10: Using the Windows Ribbon	This chapter examines the use of the Windows Ribbon control, which is designed to help users find, use, and understand available commands for a particular application in a way that's more natural and intuitive than menu bars or toolbars.
11: Using the Windows Imaging Component	In this chapter you will learn how the Windows 7 Imaging Component is used in the Hilo Browser and Annotator applications. The Windows 7 Imaging Component (WIC) allows you to load and manipulate images and their metadata. The WIC Application Programming Interface (API) has built-in component support for all standard formats. In addition, the images created by the WIC can be used to create Direct2D bitmaps so you can use Direct2D to change images.
12: Sharing Photos with Hilo	In this chapter, we'll describe how the Hilo applications have been extended to allow you to share photos via an online photo sharing site. To do this, Hilo uses the Windows 7 Web Services application programming Interface (WSSAPI). The Hilo Browser application has also been updated to provide additional user interface (UI) and touch screen features, and the Hilo Annotator application has been extended to support Windows 7 Taskbar Jump Lists. This chapter provides an overview of these new features.
13: Enhancing the Hilo Browser User Interface	In the final version of Hilo, the Annotator and Browser applications provide a number of enhanced user interface (UI) features. For example, the Hilo Browser now provides buttons to launch the Annotator application, to share photos via Flickr, and touch screen gestures to pan and zoom images. In this chapter we will see how these features were implemented.
14: Adding Support for Windows 7 Jump Lists & Taskbar Tabs	The Hilo Browser and Annotator support Windows 7 Jump Lists and taskbar tabs. Jump Lists provide the user with easy access to recent files and provide a mechanism to launch key tasks. Taskbar tabs provide a preview image and access to additional actions within the Windows taskbar. In this Chapter we will see how the Hilo Browser and Annotator applications implement support for Windows 7 Jump Lists and taskbar tabs.
15: Using Windows HTTP Services	The Hilo Browser application allows you to upload photos to the Flickr online photo sharing application. To do this, Hilo uses Windows HTTP Services. This chapter will explore how this library is used in the Hilo Browser to implement its photo sharing feature.
16: Using the Windows 7 Web Services API	The Hilo Browser application allows you to share your photos via Flickr by using the Share dialog. The previous chapter showed how the Share dialog uses the Windows HTTP Services API to upload the selected photos to Flickr using a multi-part HTTP POST request. Before the photo can be uploaded the Hilo Browser must first be authenticated with Flickr by obtaining a session token (called a frob), and then authorized to upload photos by obtaining an access token. To accomplish these two steps, Hilo Browser uses the Windows 7 Web Services Application Programming Interface (WWSAPI) to access Flickr using web services. In this chapter we will explore how the Hilo Browser uses this library.

Additional Resources

This series is targeted at C++ developers. If you are a C++ developer but are not familiar with Windows

development, you may want to check out the Learn to Program for Windows in C++ [http://msdn.microsoft.com/en-us/library/ff381399.aspx] series of articles.

Copyright Notice

Chapter 1: Introducing Hilo

The Microsoft Windows® 7 operating system provides many innovative, state-of-the-art features that support applications with a rich and responsive user experience. These features are provided through an extensive collection of libraries that give developers access to Windows 7, and to the many benefits of modern hardware. By using these Windows Application Programming Interface (API) libraries developers can create richly featured, compelling Windows applications.

"Hilo" is a series of articles and sample applications that show how you can leverage the power of Windows 7, and the Visual Studio® 2010 and Visual C++® development systems to build high performance, responsive rich client applications. Hilo provides both source code and written guidance to help you design and develop Windows applications of your own.

The series is targeted at C++ developers (if you are a C++ developer but aren't familiar with Windows development, you may want to check out the Learn to Program for Windows in C++ [http://msdn.microsoft.com/en-us/library/ff381399(v=VS.85).aspx] series of articles on MSDN). The series covers many topics, including key Windows 7 capabilities and features, the design process for the user experience, and Windows application design and architecture. Source code is provided so that you can see firsthand how the accompanying sample applications were designed and implemented. You can also use the source code in your own projects to produce your own rich, compelling Windows 7 applications. The Hilo sample applications are designed for high performance and responsiveness and are written entirely in native C++ using Visual C++.

This and subsequent articles describe the design and implementation of a set of touch-enabled Windows applications that allow you to browse, select, and work with images. They will illustrate how to write applications that leverage some of the powerful capabilities that Windows 7 provides. You will see how the various Windows 7 technologies can be used together to create a compelling user experience, and detail the design and implementation of the sample applications themselves.

The first Hilo sample application—the Hilo Browser—implements a touch-enabled user interface for browsing and selecting photos and images. You can download the code for the Hilo Browser application from the MSDN Code Gallery [http://go.microsoft.com/?linkid=9730262] ; we'll be releasing the code for the other Hilo applications soon. This article will introduce you to the Hilo Browser application and to Windows 7 application development by describing some of the tools and APIs that you can use. In subsequent articles, we'll start to drill into the details of the Hilo Browser application.

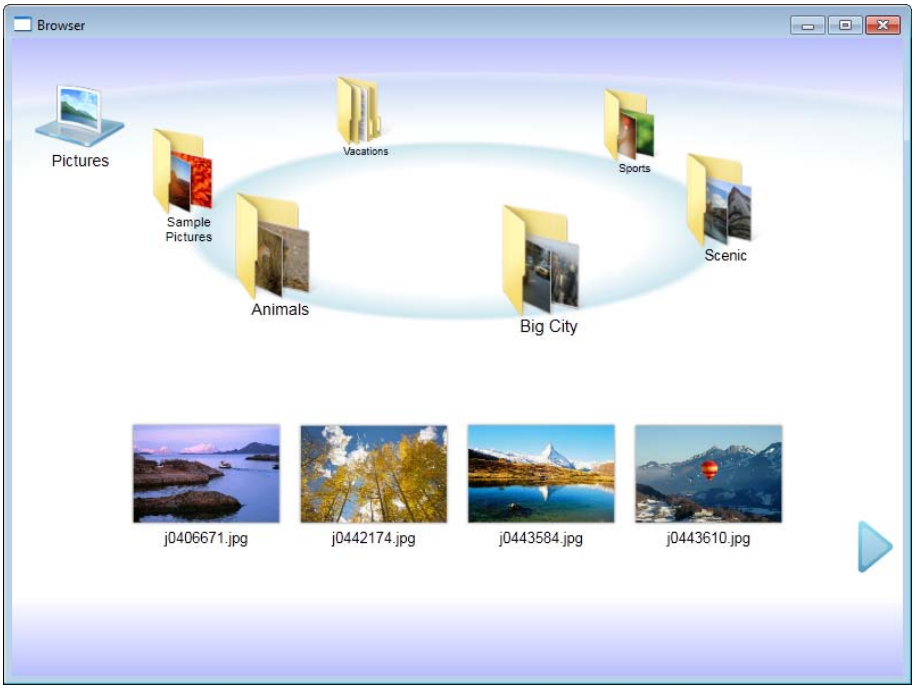
Using Hilo

The Hilo applications allow you to select, annotate and share collections of photos in a natural fashion. To do this, Hilo takes advantage of some of the key features in Windows 7: Libraries, Touch, Windows Ribbon, Direct2D, and Animation.

Tablet computers allow you to interact with applications through a touch screen and to input data through a stylus. Hilo is designed to take advantage of both the touch and pen features while still being usable through computers that only have a mouse. The animation features of Windows 7 are used in Hilo to good effect, giving you a more natural paradigm to browse files and navigate. This design was chosen so as to move away from the table-based paradigm used by most Windows applications (for example, the Tree View and List View controls), and instead to present data in a format that facilitates the use of the Touch interface while making data access simpler and more natural.

The central Hilo application is the Hilo Browser as shown in Figure 11. Figure 1 1 the **Pictures** folder has six subfolders and these are shown on the same orbital in the carousel. Through Windows 7 Touch you can rotate these folders around the orbital to bring the folder of your choice to the front. In this view the carousel will only show one complete orbital that you can spin, however, the carousel will also show partial orbitals representing the parent folders. For example, above the single, complete, orbital in Figure 1 1 the partial orbital of the **Pictures** folder.

Figure 1 The Hilo Browser showing the carousel and the media pane



The lower half of the Hilo Browser window is the media pane which shows the images in the selected folder. The media pane shows the first few images, and other images are accessible by touching (or with a mouse, clicking) the media pane arrows.

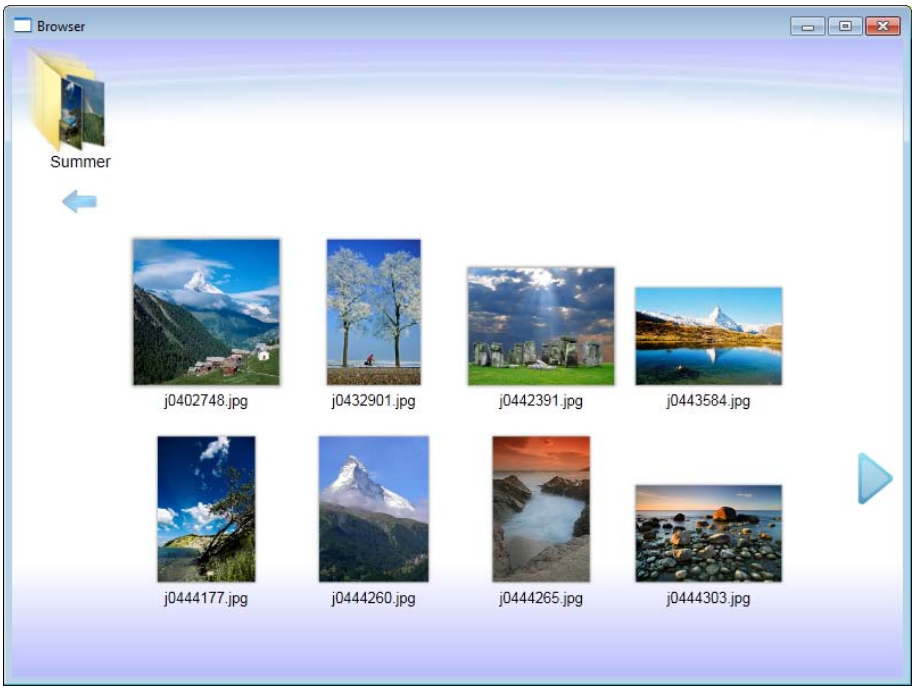
The media pane is designed for a different paradigm: touch sensitive screens. Visually, the location of a finger touch on the screen is quite imprecise, so rather than adding a small scroll bar to the media pane, in the Hilo Browser, the media pane *is* the scroll bar. The scroll bar is redundant when you can drag the items themselves using touch.

So in the Hilo Browser you have two ways to display additional items: you can either touch the scroll arrows on the left and right of the pane, or you can flick anywhere in the media pane in the direction that you want the image list to move. This is a very natural gesture, and illustrates how having a touch interface means that application designers have to change how they think users will interact with user interface (UI) items. While gestures are very natural for people accessing the UI through touch, such gestures are unnatural with a mouse, so mouse users will use the scroll arrows or the mouse scroll wheel. The Hilo Browser also allows you to scroll images using the **Page Up** and **Page Down** keys, illustrating how successful Windows 7 applications can provide input features for mouse and keyboard users as well as Touch users.

Similarly, you'll find that the rotating carousel is a natural action to perform with your finger on a touch screen. We are used to moving cards or photographs around a table, and the carousel provides a similar paradigm for accessing folders. Mouse users have full access to the carousel and they too can enjoy spinning the folders around an orbital.

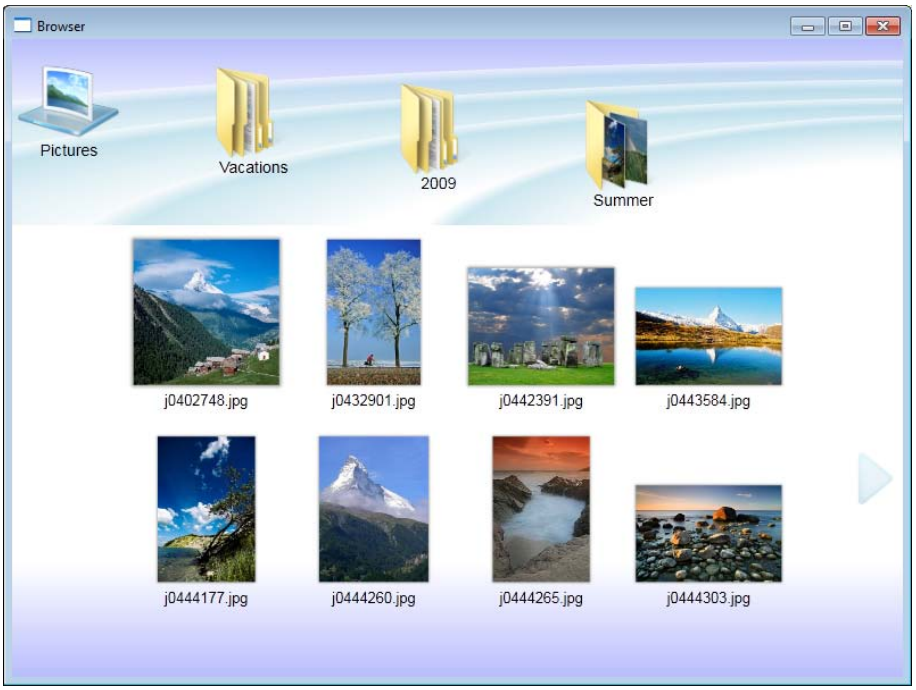
When you navigate through the folder structure, the current selected folder is placed on the stack in the top left. Figure 1 1, **Pictures** is at the top of the stack. When you select a subfolder from the carousel, for example **Summer**, this new folder is placed on the top of the stack, as shown in Figure 21 As you continue through the folder structure each parent folder is placed on the stack. The arrow beneath the stack allows you to move up a level.

Figure 2 The Hilo Browser showing the stack in the top left corner



You can expand the stack at any time through the Touch interface or via a click with the mouse. When you expand the stack all the folders are shown (Figure 31in the stack. To the right of the breadcrumb trail is the grayed out orbital of the subfolders at the bottom of the trail.

Figure 3 The Hilo Browser showing the stack expanded



The orbital of each parent folder is shown as part of the expanded stack, but only the folders that are in the breadcrumb trail are shown. Furthermore, you cannot rotate the carousel orbitals in this view. However, you can select any folder in the breadcrumb trail which gives you a quick way to move to another folder and display its subfolders in the carousel. Restricting the breadcrumb trail like this to show only the folder in the trail gives a clean and less cluttered interface than, for example, the Tree View used in Windows Explorer, and makes it much easier for you to focus on the relevant folders.

To revert back to the previous carousel view you merely have to touch or click on a folder. In this way you can expand and collapse the stack, making navigation through the folder tree a simple and natural action.

Developing Windows 7 Applications

A commercial application has to be functional and (typically) provides services that are not offered by other products. However, merely functional is not enough in a competitive market. Applications have to offer a compelling user experience and this where the features of Windows 7 become very important. The Hilo Browser takes full advantage of these features. There are four main technologies used in the Hilo Browser: Direct2D, the Animation Manager, the Touch API, and the Shell API.

Creating a User Interface with Direct2D

An important aspect of creating a compelling application is to provide a UI that is both visually appealing and practical. Modern computers have graphics cards with powerful graphics processing units (GPU) and use high resolution monitors. Direct2D is written for modern GPUs and provides access to the full capabilities of the GPUs. A Direct2D application, on a fully featured graphics card, will use hardware acceleration. However, to allow as many graphics cards as possible to be programmed with Direct2D, the underlying library checks the capabilities of the card's GPU, and if the GPU does not support a feature the DirectX® application programming interface will provide a software implementation. This means that developers write the same code regardless of the graphics cards used.

Direct2D uses a different paradigm from Graphics Device Interface (GDI) programming in previous versions of Windows. With GDI programming, the program code determines how the pixels on the screen will change, and the computer's central processing unit (CPU) performs the calculations to change those pixels. In Direct2D programming, the program indicates how the display will change, and the library tells the graphics card's GPU to do the required calculations and update the display. This frees up the computer's CPU to perform data crunching work, and results in a faster, more responsive application. Furthermore, GPUs support advanced features like antialiasing and alpha channel (opacity) blending that allows you to provide stunning user interfaces.

The Hilo Browser uses these features of Direct2D to full advantage. For example, the orbitals shown in the carousel (Figure 31are drawn with a gradient brush.

Using the Animation Manager

The Animation Manager is an API that allows developers to animate sequences of UI elements. In its simplest form, animation is a series of frames where one or more items change. The change may be the size, the color, the position or orientation of the UI item, and the change itself may be smooth or abrupt, linear or logarithmic. Constructing the series of frames into a storyboard to be played back has historically required a lot of coding and a lot of CPU processing at runtime. Furthermore, to make the animation as smooth as possible, the code must ensure that frames are displayed synchronized with the refresh rate of the monitor.

The Hilo Browser uses animation to good effect. Some animation is functional, for example the carousel displays folders by using a rotating paradigm that scales the size of the folder icon and caption according to the perceived position of the folder on the carousel. This means that the folder nearest to the user (the folder that the user is interested in) is shown larger with a larger caption than the other folders.

In other cases animation is meant to provide a compelling user experience. For example, when you touch a folder you zoom into the carousel, which means that the previous orbital expands until it reaches the existing orbitals shown at the top of the screen. The selected folder also moves to the top of the stack. At the same time the media pane is populated with thumbnails of as many images as the pane will display. The Hilo Browser displays these images, sliding in one by one from the left, as if they are playing cards being dealt on a table. This animation is more than just making the screen update more interesting; it serves an important purpose too. The human brain is more sensitive to movement than to static images, so this animation brings the user's attention to the thumbnails being shown. As a developer you have to draw the user's attention to the important data items, and the Hilo Browser shows how you can do this with animation.

Using the Touch API

The natural behavior is for users to select and move items on the screen by pointing to them with their finger or stylus. The Touch API allows you to provide this behavior to your applications, improving the user experience.

In the Hilo Browser, the user spins the carousel by dragging a folder with their finger and, as would be the case in real life, when the user removes their finger from the screen the carousel continues to spin but decelerates to a

stop. The user can stop the carousel while it is spinning by touching a folder with their finger. By building in such natural interactions with the application, the designers of the Hilo Browser make it easy for a first time user to learn.

Using the Shell API

The Hilo Browser is a visually rich application. Standard folders are displayed using the well-known folder icon. User-created folders are displayed showing a preview of the folder contents. The media pane shows thumbnails of the images in the selected folder, and each thumbnail is the same size, regardless of the size of the actual image.

All of these features are provided by the Windows 7 Shell API. This API allows you to access the shell's visual representation of items, such as folder images, as well as the icon or thumbnail for individual files The Shell, and Windows Explorer display these images, and your application can do this too, which means that your application displays are consistent with the Windows 7 user experience.

The Hilo Browser displays a thumbnail for the images in a folder. There are many image types and so the Hilo Browser must be able to extract the image information and create a thumbnail in a bitmap format that can be displayed on screen. To do this, the Hilo Browser uses the Shell API to obtain the image information from a file, and it then uses the Windows Imaging Component (WIC) to convert the image into a bitmap that can be displayed with the Direct2D API. WIC has support to load and manipulate images for all the standard image formats (TIFF, GIF, JPG, PNG, ICO, BMP) and metafile formats.

Furthermore, since loading and processing images is time consuming, there is a possibility of making an application sluggish. This is not the case with the Hilo Browser. Partly this is due to the clever use of animation when the images are displayed in the media pane, but the main reason is that the Hilo Browser provides code that asynchronously loads images on a background thread. The Hilo Browser can do this because Direct2D is multi-threaded aware. As a consequence the Hilo Browser is responsive in spite of the image processing that it has to do.

Developing the Hilo Browser Application

The Hilo Browser application is the first in a series of sample applications that allow you to browse, select, and work with images. You can download the code for the Hilo Browser application from the MSDN Code Gallery [http://go.microsoft.com/?linkid=9730262] . We'll be releasing the code for other Hilo sample applications soon.

The next articles in the series will describe the design and implementation of the Hilo Browser application in detail. Before then though, it's useful to review the high-level design decisions behind the Hilo Browser application. These decisions include the choice of development language, application framework, design paradigm, and tooling.

The rich user experience of Windows 7 is best accessed through a powerful, flexible language, and that means C++: by using C++ you can access the raw power of the Windows 7 API. To build the Hilo sample applications, all you need is Visual C++ Express and the Windows 7 SDK, both of which are available as free downloads.

Hilo applications show how to design and develop an application for Windows 7. But while the code showcases the Windows 7 APIs, it is not wedded to any particular application framework. Instead, Hilo implement a lightweight common application layer that directly uses and highlights the Windows 7 API rather than obscuring it. This common application layer is used to support all of the Hilo applications. It illustrates the best practices for developing Windows applications, and while it is not complete—it was designed simply to provide the features needed by the Hilo applications—it does show the best practices used in designing re-usable frameworks and can be extended to provide additional features.

The key design decisions behind the code for the Hilo Browser are simplicity and readability. That is the main reason why the common application layer is lightweight, but it also means that error checking and tracing is kept to a minimum. This is not because errors and tracing are unimportant (in fact they are vital aspects of any code design) but because extensive use of the tracing will distract from the main aim of showing how to use the Windows 7 APIs.

The Hilo Browser is an object orientated C++ design using the best practices of interface programming, abstraction and object factory patterns. The design is also chosen to give a responsive user interface and to this end, uses asynchronous data handling through a background thread. You can use all of these classes in your own projects.

Lastly, application development requires a powerful and fully featured development environment. Visual Studio 2010 offers an integrated environment where you, as a developer, can use your existing skills to code and debug your projects. Visual Studio provides powerful tools to help you deliver quality code quickly. There are four editions delivering varying levels of tool support which means that you can pay for the level of tool support you need.

Conclusion

This article gave an introduction to the Hilo Browser application. It showed you the features of the Hilo Browser and introduced the key Windows 7 technologies used to provide those features. The next article will cover how to set up your development environment, and explains how to install Visual C++ and the Windows 7 SDK so that you can compile and run the Hilo Browser source code.

Chapter 2: Setting up the Hilo Development Environment

The Hilo Browser application is a sample application written in C++. The application uses the Microsoft Windows 7 operating system application programming interface (API) through the Win32 and DirectX libraries. This article outlines how to set up a workstation for the development environment so that you can compile and run the Hilo Browser sample application. In subsequent articles, we'll be walking through the code and explaining the rationale for the choice of technologies, the UI design, and the implementation of the application's features.

Setting up the Developer Workstation

Windows 7 offers a wealth of Windows C/C++ API features for developers. The Hilo Browser application has been developed using Visual C++ 2010 Express. The minimum machine configuration needed to install Visual C++ 2010 Express is:

- 1.6 GHz or faster processor
- 1024 MB RAM (1.5 GB if running on a virtual machine)
- 3 GB of available hard-disk space
- 5400 RPM hard-disk drive
- DirectX 9-capable video card running at 1024 x 768 or higher display resolution

The Windows Ribbon will be used in subsequent Hilo applications. To write code for the Ribbon you must install the Windows Software Development Kit (SDK) for Windows 7. The SDK requires a minimum of 2.5 GB of free space, so this means that to install the software needed to compile all of the Hilo applications you need at least 5.5 GB of disk space. The installation of the SDK is covered in detail below.

Clearly a free development environment will have some restrictions, but perhaps surprisingly, the restrictions are not great. Visual Studio 2010 Express editions give the developer a functional development environment that can be used to create fully featured applications.

The features of the various Visual Studio editions are available on the MSDN website [http://msdn.microsoft.com/en-us/library/hs24szh9(VS.100).aspx] . Compared to the Professional edition, the main features that the Express edition lacks are that it only provides the 32-bit compiler, and it lacks the ActiveX Template Library (ATL) and Microsoft Foundation Classes (MFC) libraries. The C++ compiler is fully featured except that it does not provide profile guided optimizations. Furthermore, the Express edition does not supply all the tools provided with the Professional edition, but perhaps the one that you will miss early on is integrated resource editor—you will have to use external editors to edit bitmaps and icon files (Windows 7 Paint will do this).

Downloading Visual Studio from the MSDN Website

Visual C++ 2010 Express is available as a free download using the web download tool from the following address:

http://www.microsoft.com/express/Downloads/#2010-Visual-CPP
[http://www.microsoft.com/express/Downloads/#2010-Visual-CPP]

Once downloaded you should run the setup tool (Figure 1), agree to the license conditions, and select whether to download the optional components. Hilo does not require the optional components (Silverlight nor SQL Server 2010 Express) so deselecting these options will make the download and installation quicker (Figure 2).

Figure 1 The web download and installation application

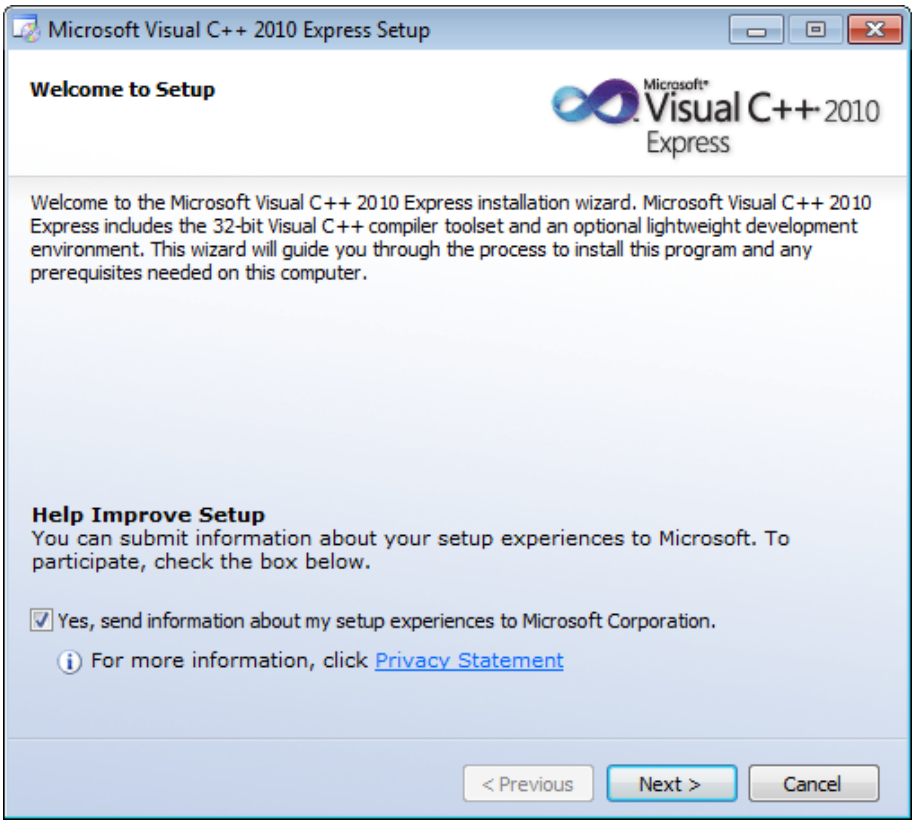
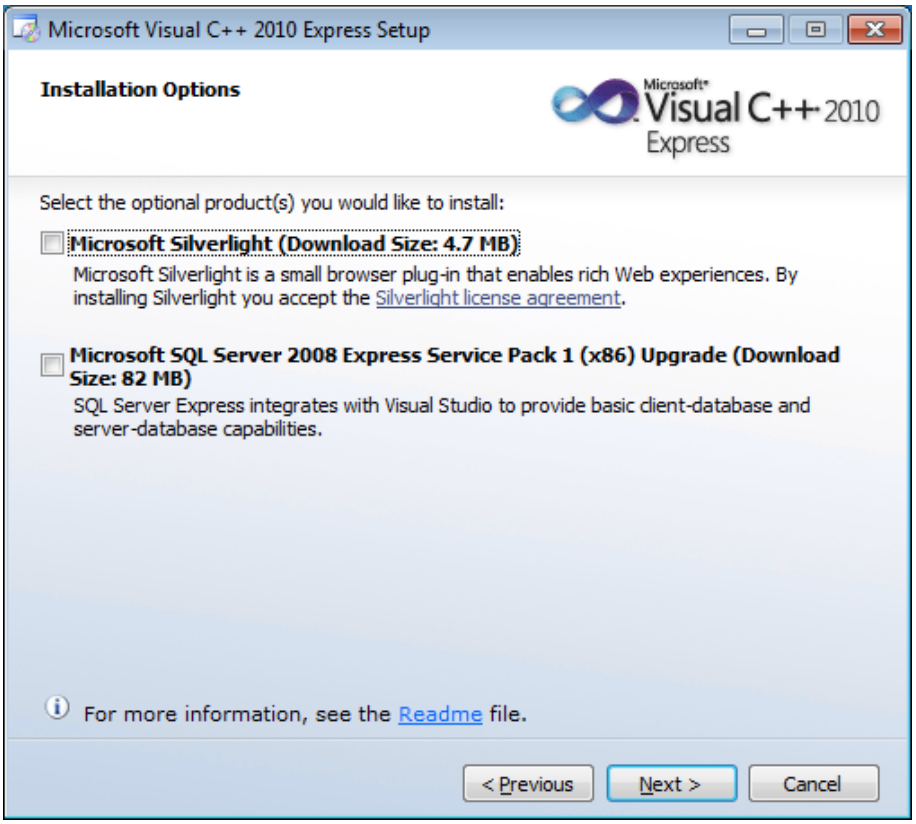
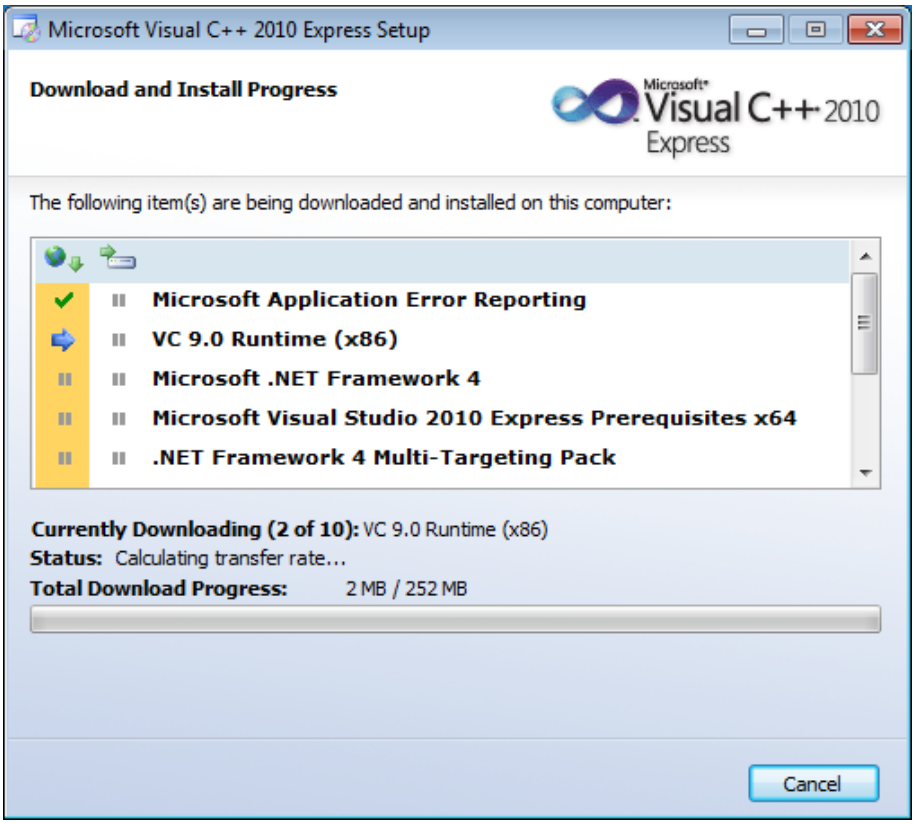


Figure 2 The download options



Next you will be asked where to install the application (here you should use the defaults) and finally you will be shown a progress dialog (Figure 3) showing the components being downloaded and installed. As a guide, the minimum installation of Visual C++ will involve downloading 146 MB of data.

Figure 3 Downloading Visual C++ 2010 Express

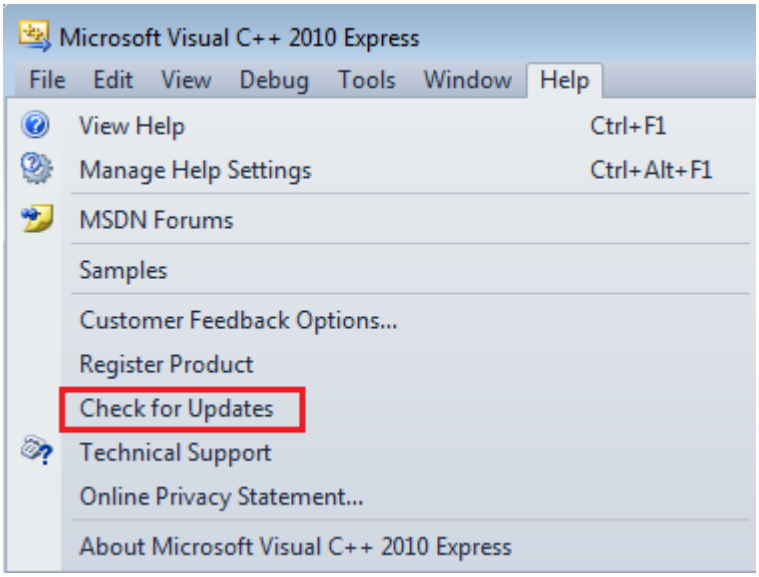


Although the Hilo application does not use SQL Server, you will notice that Visual C++ 2010 will download and install SQL Server Compact edition. This is a separate product and it is used by Visual C++ 2010 to create the C++ program database (in previous versions of Visual Studio this was the .ncb file, which is used to store IntelliSense data for your code).

Registering Visual C++ Express Edition

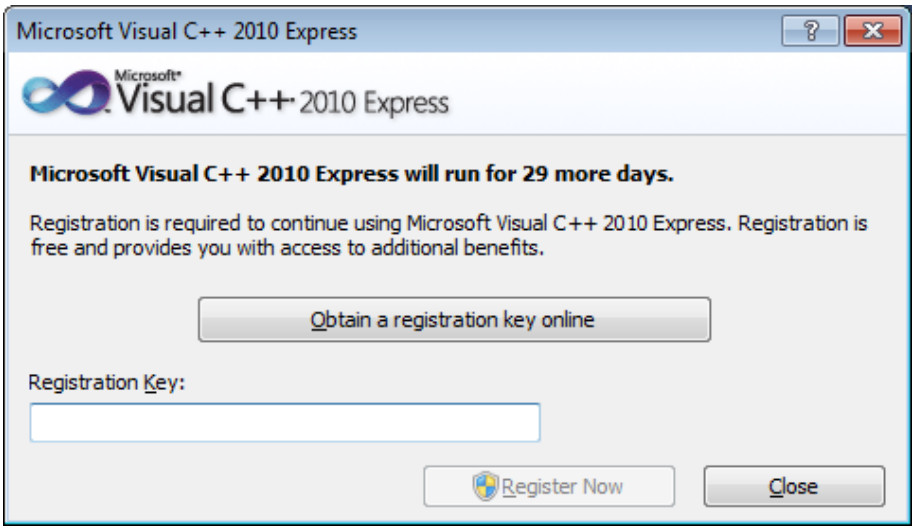
Once Visual C++ 2010 Express has been installed you may run it for 30 days before the trial period expires. To use it beyond this time you need to register it. The registration option is on the Help menu (Figure 4).

Figure 4 Command to register Visual C++ 2010 Express



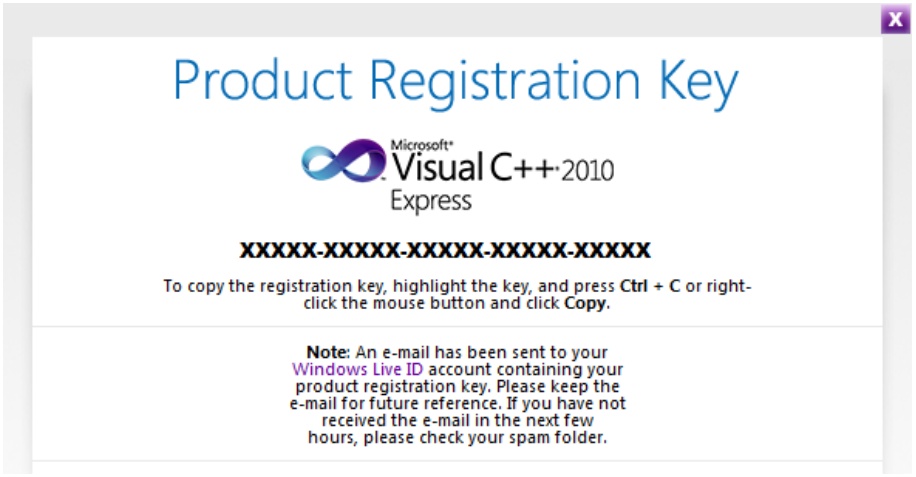
The menu command gives the registration dialog shown in Figure 5.

Figure 5 Registering Visual C++ 2010 Express



When you click on the button marked **Obtain a registration key online** your browser will open the registration page on the Microsoft website. To obtain a registration key you must first log on using a Windows Live ID (which requires registering some minimal details) and then the website will return a registration key.

Figure 6 Obtaining a registration key

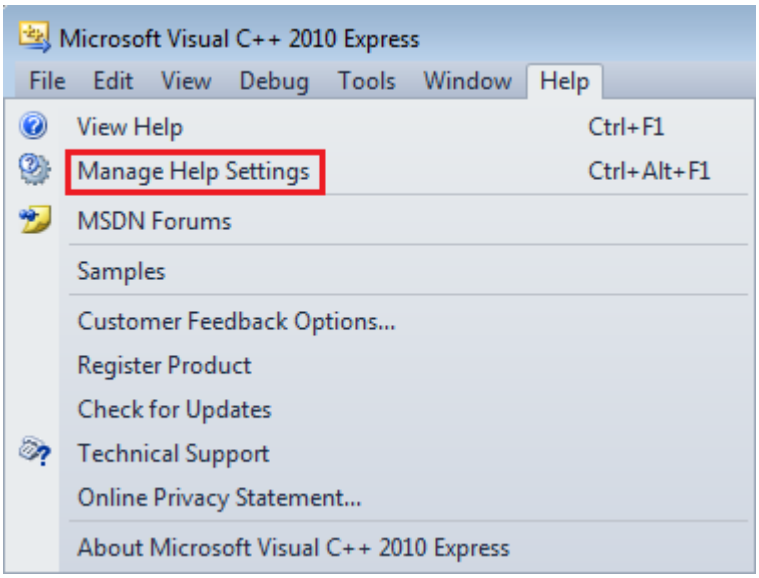


You should copy this key and paste it in the **Registration Key** box on the registration dialog (shown in Figure 5) and then click the **Register Now** button. Once you have registered the product the time restriction will be removed.

Using Help

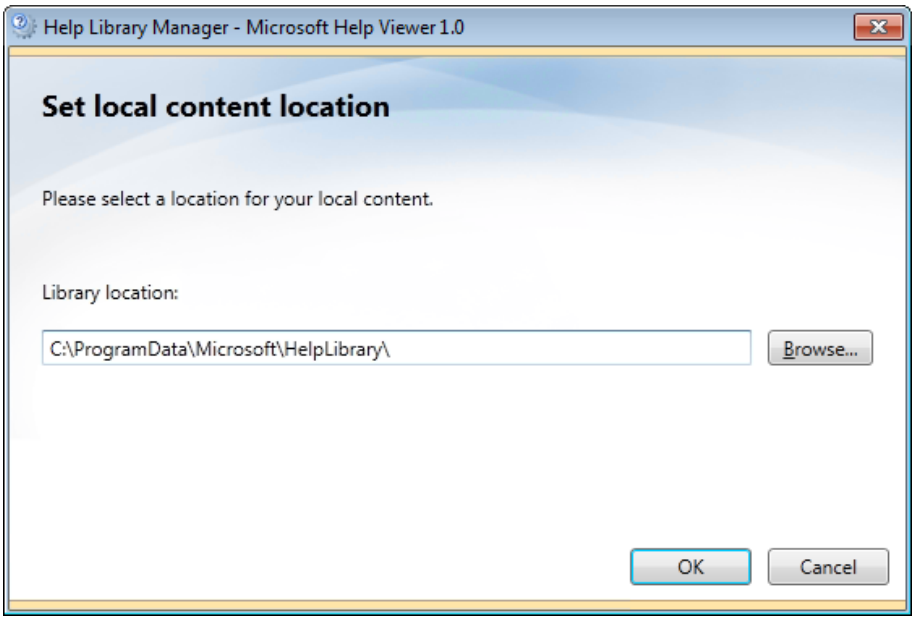
One of the most important parts of a development environment is the help system. Visual C++ 2010 comes with two types of help documentation: local and online and these are configured through the **Manage Help Settings** item on the **Help** menu (Figure 7).

Figure 7 Managing the source of the help documentation



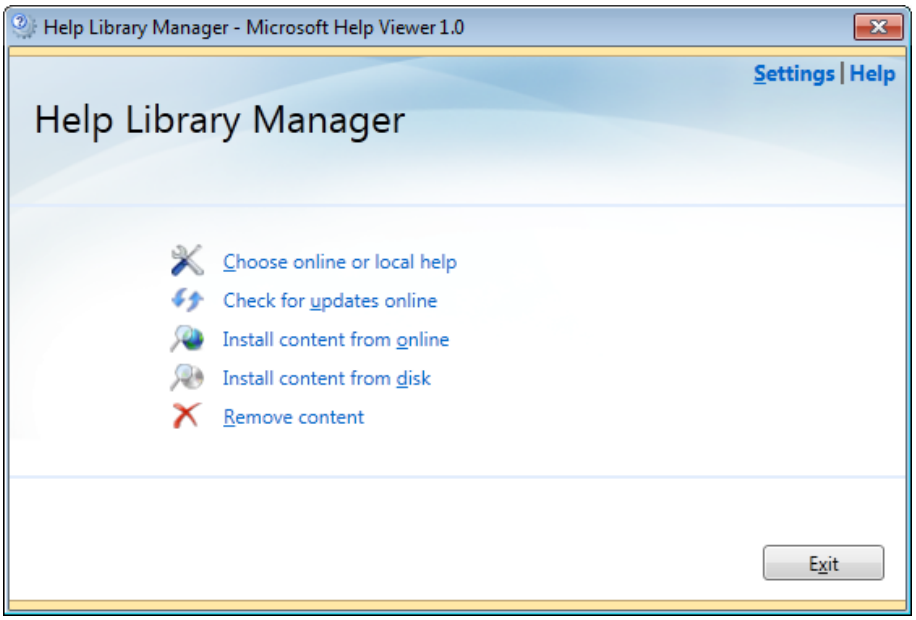
The first time that you run this command you will be asked for the location of the local help system (Figure 8), accept the value suggested by clicking **OK**.

Figure 8 Starting the Help Library Manager



Next you will see the actual **Help Library Manager** (Figure 9).

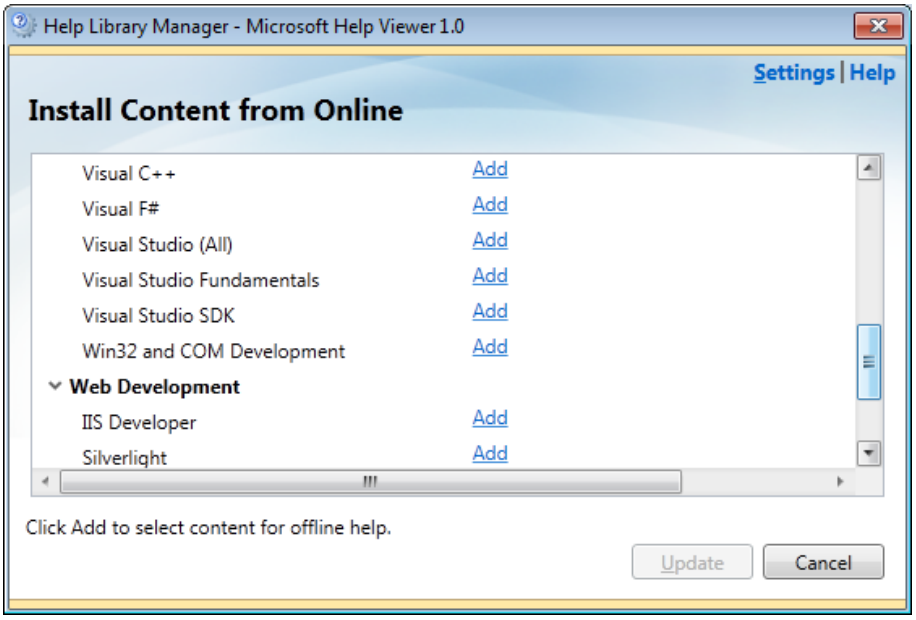
Figure 9 Running the Help Library Manager



Visual Studio 2010 help is browser based, but you have to choose whether the help data is locally installed, or web based. If you select the option to use the online help system then you can use help immediately. The advantage of the online system is that each page has a **Community Content** section that may contain helpful comments from members of the online community, and indeed, you can even add your own tips.

The local help system does not benefit from the community contributions, but since it is installed on your computer it means that you can use it even if you do not have a network connection. Before you can use local help you must install it. The help content is supplied as libraries of related APIs and to download and install this content you should use the Help Library Manager option **Install content from online**. When you select this option the manager will first search the web for available content and list the help topics in a table with an **Add** link in the **Action** column next to each help topic (Figure 10).

Figure 10 Installing documentation from online



The following help topics are relevant to Visual C++ Express:

- Visual Studio C++
- Visual Studio (All)
- Visual Studio Fundamentals
- Win32 and COM Development

When you have selected at least one topic you can click the **Update** button to download and install the documentation. This process may take a while (if you choose the libraries in the list above, about 1 GB will be downloaded) so now is definitely a good time to have a cup of coffee or two. Unfortunately the Manager does not warn you before the download starts how much disk space will be needed, so you should ensure that you have several gigabytes of free space to take into account the space needed for the final documentation and for the intermediate files. Once the Manager informs you that the documentation has been installed you can close the main Manager dialog by clicking **Exit**.

The first time that you use the locally installed help system Visual C++ 2010 will start the Help Library Agent. This is a simple HTTP server running on port 47873 on your computer. When you press F1 in Visual C++ 2010 to obtain help for a feature or a keyword the help system will determine the web page and create a URL to the appropriate file on port 47873 on the loopback address. The Help Library Agent will then locate the page within the compressed help files and return them to your registered browser.

Other than the community content section, there is no difference between the online and local help. Both help systems have a search box and show a tree view of the available help (to aid browsing).

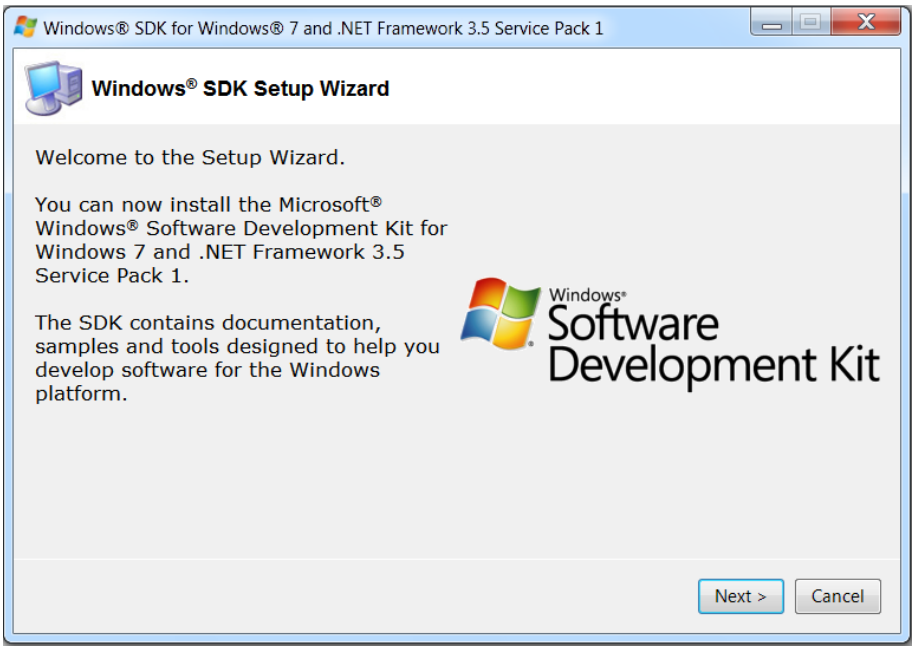
Installing the Windows 7 SDK

There are two ways to install the Windows SDK for Windows 7 through the web tool or by downloading an ISO image of the DVD and then installing the SDK from the DVD. The DVD ISO image contains the entire SDK, but at install time you can choose which items you install. Both the web tool and the ISO image are available from the Microsoft download site.

The web tool is called winsdk_web.exe [http://www.microsoft.com/downloads/details.aspx?FamilyID=c17ba869-9671-4330-a63e-1fd44e0e2505&displaylang=e] and is has an initial download size of 492 K. There are three ISO images available on the download site [http://www.microsoft.com/downloads/details.aspx?familyid=71DEB800-C591-4F97-A900-BEA146E4FAE1&displaylang=en] , for x86 32-bit, AMD64 64-bit and for Itanium 64-bit development. The Express edition of Visual C++ 2010 is only available in a 32-bit version, but you still must install the version of the SDK for the platform where you install Visual C++. The ISO files are approximately 1.5 GB in size.

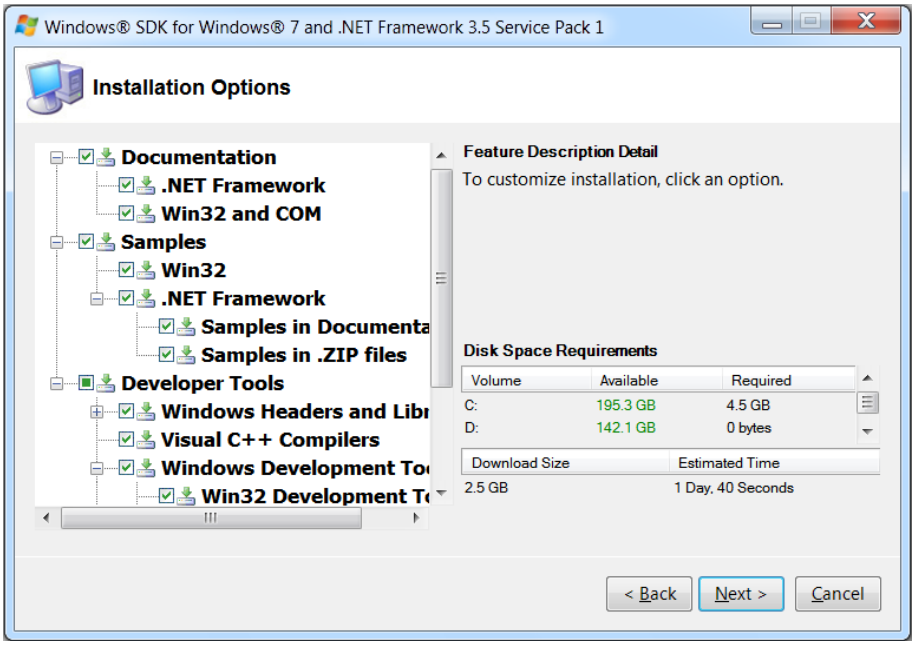
Once you have burned the DVD you can start the installation by running the **setup.exe** file in the root folder of the DVD. Alternatively, you can run the web tool **winsdk_web.exe**. Both tools have the same user interface and start with an introduction page, Figure 11.

Figure 11 The Windows SDK installation program



The following pages request that you accept the license agreement and then provide the location to install the SDK, accept the defaults. The next page lists the items that can be installed, Figure 12.

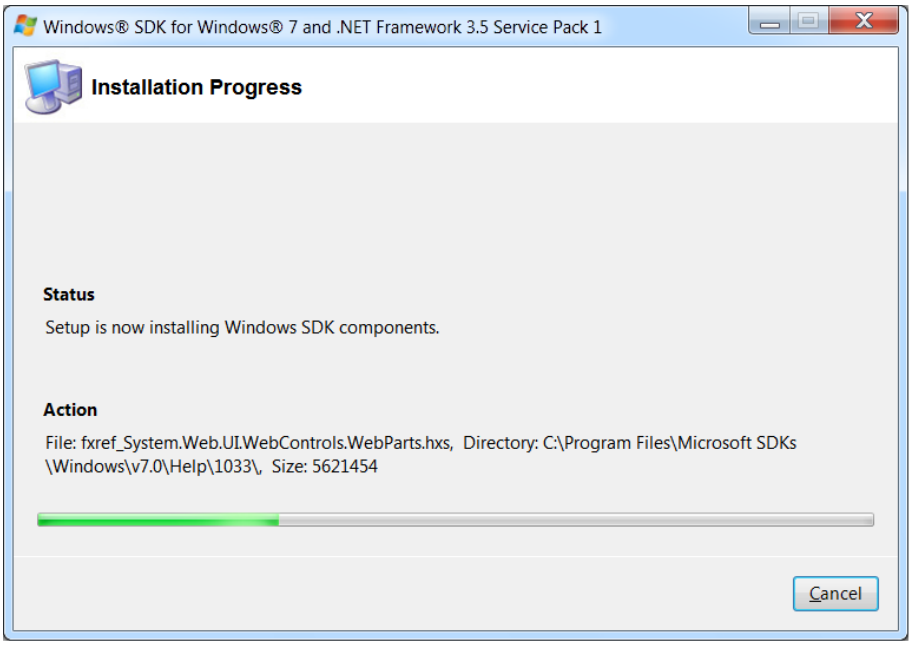
Figure 12 Installation options for the Windows 7 SDK



The tool indicates the disk space required for the selected options and, if you are using the web tool, you will also see an estimate of the amount of data that will be downloaded. A full install requires 4.5 GB of disk space and will download 2.5 GB of data. If you have sufficient disk space and installing from the DVD then you can simply install the entire SDK. If you have limited disk space or you are installing from the web then the absolute minimum you should install is the **Win32 Development Tools** from the **Development Tools**, **Windows Development Tools** section. The **Win32 Development Tools** will take up 42 MB of disk space.

Once you have selected the install options and clicked the **Next** button the setup program will download, decompress and install the items you selected, Figure 13.

Figure 13 Installing the SDK



When the setup program has finished you must reboot the computer to complete the installation. You do not need to make any changes to your C++ projects since Visual C++ 2010 will automatically add the paths to the SDK tools to the search path it uses.

The SDK uses the Microsoft Document Explorer to display help, rather than the web browser based system used by Visual C++ 2010. The Microsoft Document Explorer is the help system used by previous versions of Visual C++, however, it is not integrated into the Visual C++ 2010 development environment.

Configuring Visual Studio

All of the settings of the Visual C++ 2010 Express environment can be changed. This allows you to configure the environment for your own preferences or to your employer's coding standards. The main configurations are accessed through the **Options** menu item on the **Tools** menu. This dialog gives five categories of configuration options shown in the following table.

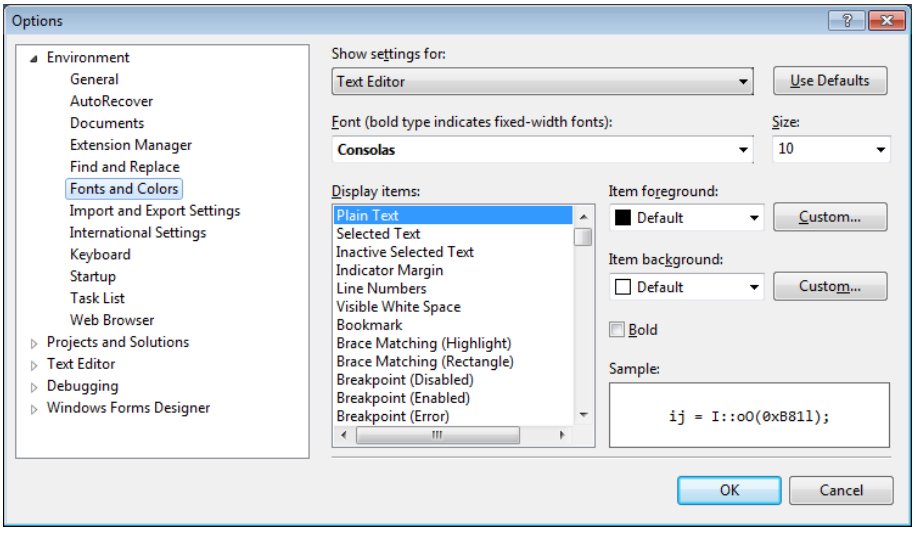
Category	Description
Environment	General options for the integrated development environment, things like the fonts that will be used, keyboard short cuts, and startup options.
Projects and Solutions	Options for the general locations of solutions, how and when projects are built, and information about things like the extensions of C++ project files.
Text Editor	Options for the size of tabs and whether tab characters or spaces are used, and options about indenting code.
Debugging	General options for how the debugger works, whether Edit and Continue is allowed, and the location of symbol files.

There are too many options to list here in detail, however, every developer is different and customizing the development environment to something that the developer is comfortable with is important, so we will list the common settings here.

Changing the Fonts

The **Environment** category has a subcategory called **Fonts and Colors**, Figure 14.

Figure 14 Configuration page used to change the code editor fonts

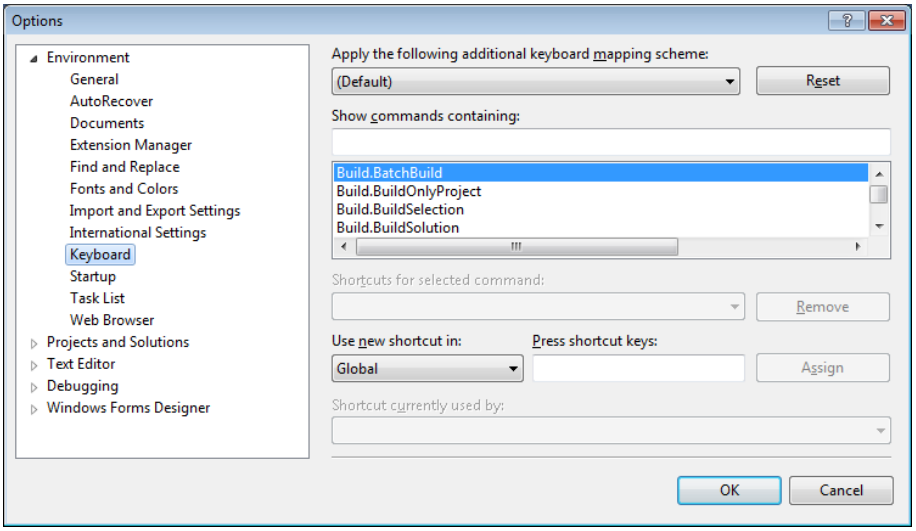


This page allows you to change the font used for individual tool windows, or several related windows within the design environment. Through this dialog you are able to choose a font size and font type that you are comfortable using. It is perhaps one of the most important configuration settings because you will spend most of your time reading text.

Changing the Keyboard Shortcuts

The **Environment** category has a subcategory called **Keyboard**, shown in Figure 15.

Figure 15 Configuration page used to change keyboard shortcuts



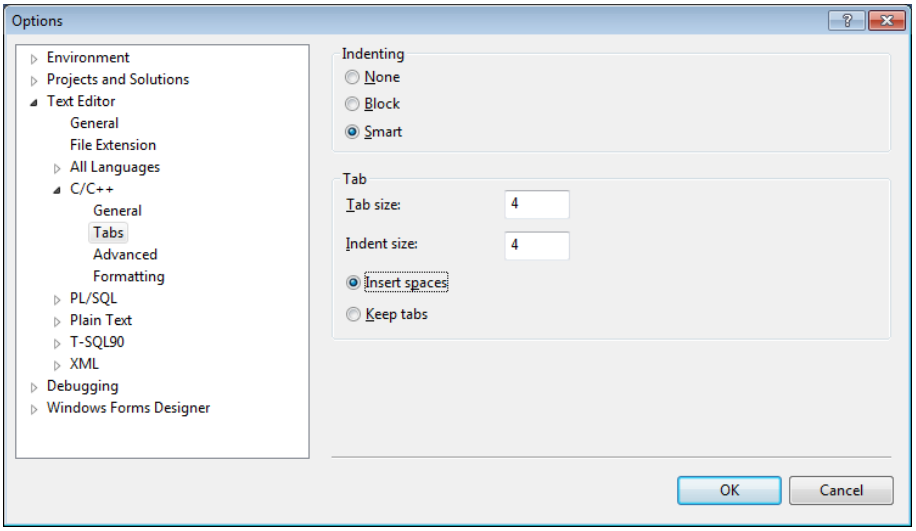
This page allows you to specify the keyboard shortcut keys that you can use to perform commands in the development environment. The commands mentioned in the list box are menu items. To change a command you click on the **Press shortcut keys** text box and press the appropriate key combination, the page will inform you if this key combination is already being used and allows you to choose to assign the key combination to the command.

If you typically use another development environment you may be used to different keyboard shortcuts than the defaults for Visual C++ 2010 Express. The ability to change the keyboard shortcuts means that you can get started to using Visual C++ immediately without having to learn new keyboard combinations.

Changing the Code Editor Settings

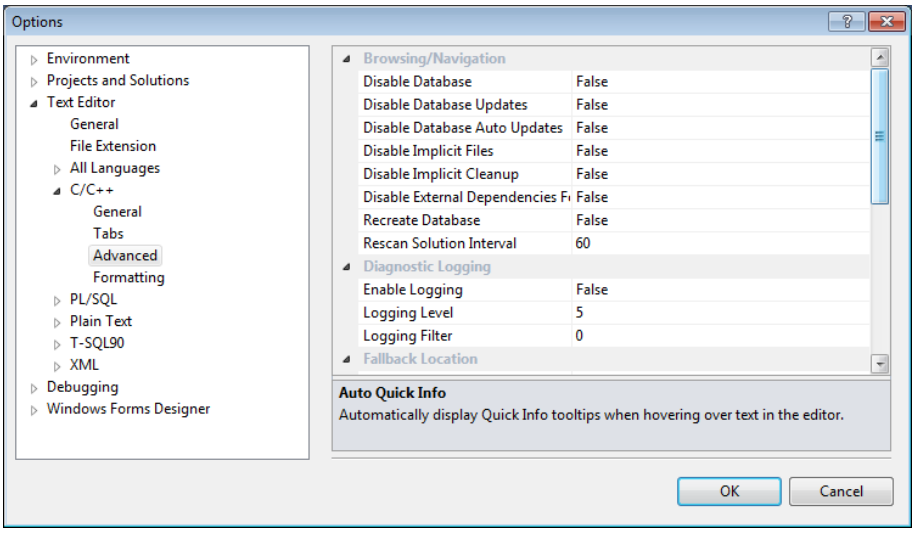
Code formatting is sometimes personal preferences, and sometimes a company policy, so it is important to know how to change them. Figure 1 shows the **Text Editor** category that you can use to change the tabs and indentation for C++ code.

Figure 16 Text editor tab options



The **Advanced** options are about the database that will be created to enable code browsing, shown in Figure 17.

Figure 17 The code editor Advanced options

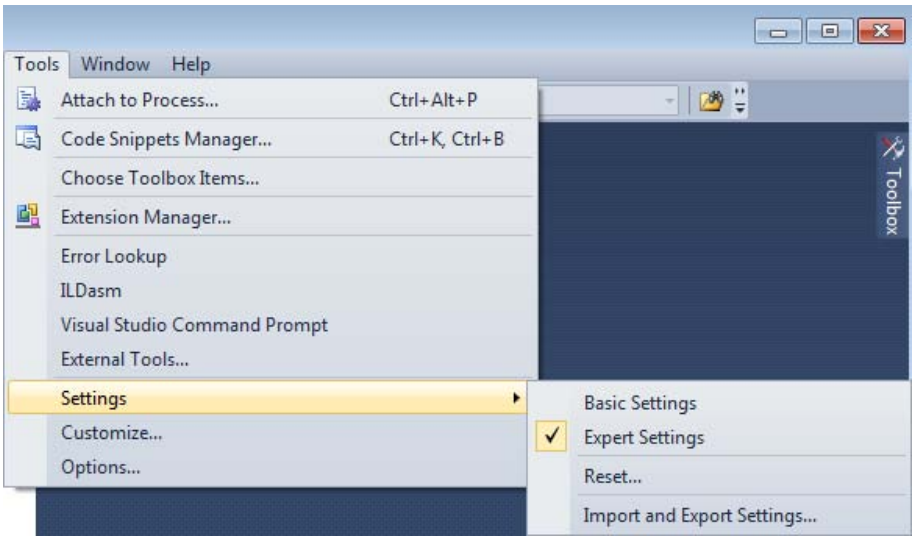


For example, the **Rescan Solution Interval** (by default 60 minutes) indicates how often the solution is rescanned to refresh the database. The browser database contains information about the classes and global functions in the project and this information is used by **Class View**, the navigation bar at the top of code windows and by the search menu items like **Go To Definition**, **Go To Declaration** and **Find All References**. The term database is used figuratively and literally because Visual C++ Express uses a SQL Server Compact database (with the extension sdf) to hold this code browser information.

Changing Other Settings

The **Class View** is a standard tool window in previous versions of Visual C++, however, when you start Visual C++ 2010 Express it will appear that this tool is no longer available. This is not the case, **Class View** is still supplied, but you have to use the environment in Expert mode. Figure 18 show the menu items that allow you to select Expert mode. On the **Tools** menu click the **Settings** submenu and then click **Expert Settings** to show **Class View** or **Basic Settings** to hide it.

Figure 18 Menu setting to select Expert mode



In addition to exposing the **Class View**, selecting **Expert Settings** also enables other items, including the **Property Manager**, exposes menu items that allow you to enable the **Extension Manager**, and give you access to external tools like the MSIL Disassembler or **Ildasm** (for managed projects) and the **Error Lookup** utility. The **Property Manager** allows you to share project configurations between projects, while the **Extension Manager** gives you access to the installed extensions and allow you to browse the Microsoft online extensions gallery and download extensions.

Installing the Hilo Solution

The Hilo Browser solution is available as a ZIP archive from the MSDN Code gallery [http://code.msdn.microsoft.com/Hilo] . On the Downloads page click the link for the code you wish to download and accept the license agreement. The ZIP file will then be downloaded to your hard disk. Once the download has completed, unzip the contents of the archive to a folder on your hard disk (for example, the Documents\My Documents\Visual Studio 2010\Projects library). Now browse the solution folder and double click on the Hilo.sln file to start Visual C++ 2010 Express and load the solution. For each project that is loaded you may see a security dialog, click **OK** to continue loading the project.

When the projects have been loaded, you will see at the bottom of the window the status message "Parsing included files." This indicates that the environment is scanning the files in the solution and building the browsing database. When the scan has completed you can use the **Class View** and other tools like the **Find All References** context menu item.

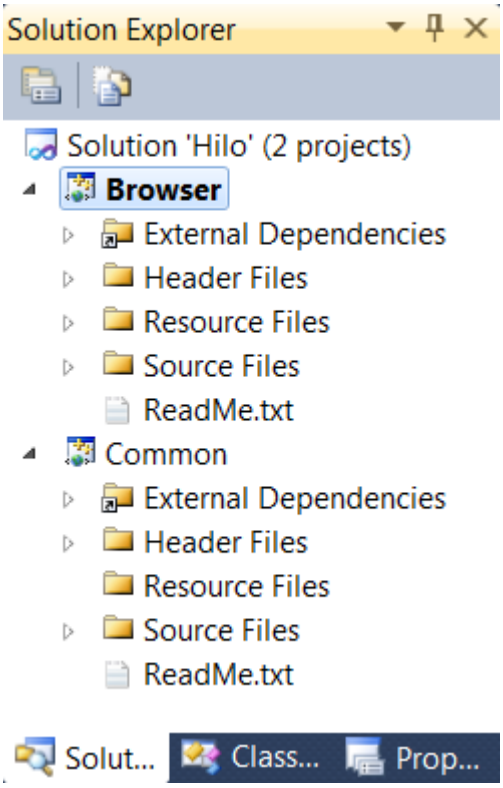
To build the solution, click the **Build** menu and then click the **Build Solution** menu item. The results of the build operation will be shown in the **Output** tool window. If the installation of Visual C++ 2010 Express and the Hilo solution was successful then the solution should build without any errors or warnings. If you're using Visual C++ 2010 Express, however, you will receive a warning about the x64 build because Visual C++ 2010 Express does not support 64 bit projects. You can safely ignore this warning.

To test the application, click the **Debug** menu and then click **Start Without Debugging**. The Hilo Browser application will start and show the contents of the Windows 7 Pictures library. You can now try out the application, spin the carousel to view the subfolders and touch (or with a mouse, click) on a folder to open it and view the images. Close the Hilo Browser when you have finished exploring its features.

Examining the Hilo Project

A Visual C++ 2010 project is made up of project files, resource files, and code files. The standard view in the **Solution Explorer** shows the code and resource files categorized as **Header Files**, **Resource Files**, and **Source Files**. **Solution Explorer** is shown in Figure 19. However, these **Solution Explorer** folders do not reflect the folders in the project disk folder, you can create any folders you like and move files between them without affecting where the file is saved.

Figure 19 Solution Explorer in Visual C++ 2010 Express



The resource files are items like icon files and bitmaps and the resource compiler script file (.rc file). The Express edition of Visual C++ does not contain the resource editor, so to change the resources that will be bound to the

executable file you must edit the resource script manually. To do this, right click on the resource script in the **Solution Explorer**, and click **View Code**. The other resources, like bitmaps and icons, must be edited with external tools, to do this double click on the item in the **Solution Explorer** and Visual C++ 2010 Express will launch the editor registered with Windows to open the file type.

On the disk, the Hilo Browser project is a subfolder of the solution folder and headers, source code files, and resource files are stored in this folder. The Browser project folder also has the project file, .vcxproj, a filters file, .vcxproj.filters and a user settings file, .vcxproj.user.

Visual C++ 2010 uses the MSBuild build tool. The .vcxproj file lists the targets in the build and the compiler and linker options used by the compilers. If you run the MSBuild utility from the command line in the project directory it will automatically load the .vcxproj file and build the targets specified in it.

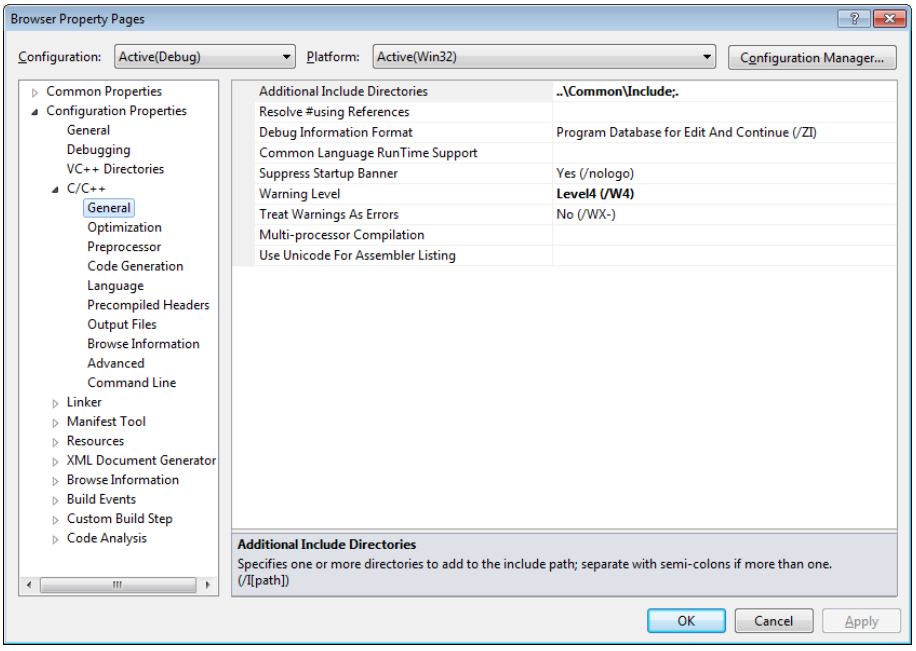
The .vcxproj.filters file lists all the targets and the folder where they appear in the **SolutionExplorer**. This file is only used by the **Solution Explorer** and it is not used as part of the build. Similarly, the .vcxproj.user file contains user interface information for the current user.

When you open the solution for the first time in Visual C++ Express the environment will parse the source files and generate the code browser database, the .sdf file, and a folder called ipch which contains the IntelliSense pre-compiled header files that are used to provide IntelliSense information as the developer types code. This means that if you intend to distribute the solution, the absolute minimum collection of files that you should include are the code and resource files, the .sln file, and the project files .vcxproj and .vcxproj.filters, all the other files will be generated by Visual C++ if they do not exist.

Changing Project Options

The options in the .vcxproj file are edited using the property pages of the **Solution Explorer**. To do this, right click on the project entry in the **Solution Explorer** (make sure that you click on the project not the solution) and from the context menu click **Properties**. These property pages are categorized into pages for the tools, build events and debugging. The project property pages are shown in Figure 20.

Figure 20 Project configuration property pages



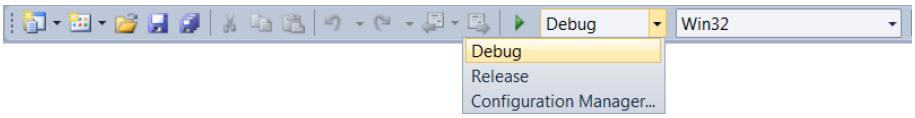
The four main tools used to build a project are in one of the following categories: **C/C++** compiler, **Linker**, **Manifest Tool**, and **Resources** for the resource compiler. In all case you will have one or more property pages where you can change individual options, and a page called **Command Line** where you can type the command line switches for the options you want to use. The build tool values that you set through the project properties will be applied to all source files, if you wish to apply an option for just one file then you should use the property pages of that individual file.

In addition to the build tool options there are also property pages for build events and debugging, and a property page called **VC++ Directories** that allows you to provide additional search folders for the current project for the include folder, library folder and executable folders.

Debugging the Solution

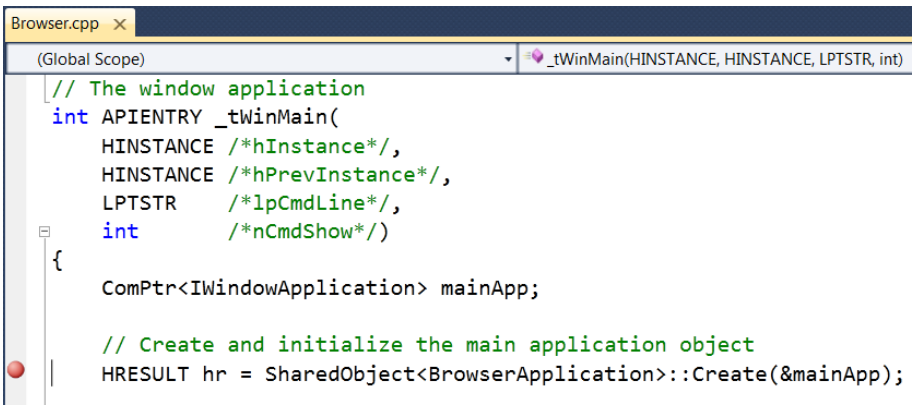
Debugging a solution with Visual C++ 2010 Express is very simple. The first action to ensure that the solution is built with debugging information, and to do this you must ensure that the **Debug** build option is selected on the **Standard** toolbar, Figure 21, and then rebuild the solution.

Figure 22 Selecting the Debug build option



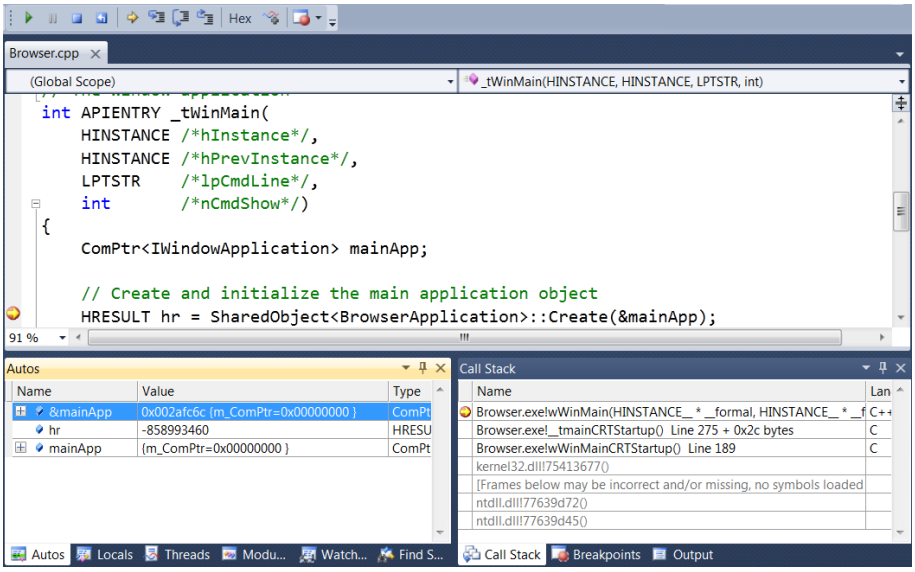
Setting a breakpoint is simple: click on the line where you wish to set a breakpoint and then through the **Debug** menu click **Toggle Breakpoint**. A red breakpoint glyph appears in the indicator margin on the left of the line, as shown in Figure 22. You can also toggle a breakpoint by clicking in the indicator margin.

Figure 22 Toggling a breakpoint in the code editor



Once you have set breakpoints you can start debugging by clicking the **Start Debugging** option on the **Debug** menu. When a breakpoint is reached the debugger will pause at that point, Figure 23, and display a yellow arrow glyph above the breakpoint.

Figure 23 Debugging the Hilo Browser



In addition several new tool windows will appear. The **Autos** window shows the values of the variables on the current line of code, the **Locals** window shows all the variables in the current scope. If the variable is a pointer to an object instance then you can expand the entry to show the values of the object's members. You may also change the value of a variable through one of these variable windows.

When the application runs under the debugger the **Debug** toolbar will show (the toolbar is shown at the top of Figure 23). This toolbar allows you to control the debugging. Using this toolbar you can run the application until the next breakpoint, stop the application or single step. If you choose to single step then you can choose to step into the function at the current execution point, or step over it as a single action, or indeed, you can step out of the current function as a single action.

Conclusion

Now that your development environment is set up and you have compiled and run the Hilo Browser application, we're ready to start walking through the code in more detail. In the next article, we'll discuss the technologies that were used to develop the Hilo Browser application.

Chapter 3: Choosing Windows Development Technologies

There are several factors to consider when choosing the right set of tools and technologies to use when developing an application. Clearly the requirements of the application are important to consider, as are the skills of the development team and the time available for developing the project. However, just as important to consider are the flexibility and capabilities of the programming languages, development tools, and libraries that will be used to implement the application. The design criteria of the Hilo applications are that they must be lightweight and high performance, yet be compelling fully-featured Windows 7 applications with minimal installation requirements. This article describes the rationale for the choice of the development technologies used to implement the Hilo applications.

Choosing the Right Language

There are many different languages to choose from, each with their own advantages and disadvantages. The first decision to make is whether to use a .NET-based language. While .NET-based languages are getting more sophisticated, they do lack the power and flexibility of more established languages. Any code written in a .NET-based language will access the operating system features through one of the .NET Framework libraries or through the .NET Platform Invoke or Component Object Model (COM) interop layer. Although these libraries and interop layers have been designed to be as efficient as possible they will always add extra machine cycles to any call. In an application that needs the highest performance possible, these extra layers will make a difference.

When you don't want to use a managed runtime environment to develop on the Microsoft Windows platform there are few candidates. The best candidates are C and C++. C gives the developer an experience that gives the developer a relatively simple, powerful programming language, but it lacks the benefits of higher level languages like encapsulation, abstraction, and inheritance. Furthermore, much of the Windows 7 Application Programming Interface (API) is accessible only through a COM interface. COM interface pointers are essentially pointers to virtual function pointer tables, and the double dereferencing that is required to access a COM method make the C code complicated and difficult to read.

So overall C++ is a good choice for writing Windows 7-based applications. It has the correct balance of the raw power and flexibility of C, and the sophisticated language features of C#. C++ as a compiled language produces executable code that is as close to the Windows API as is possible and, indeed, the C++ compiler optimizer will make the code as efficient as possible. The C++ compiler has many of options to allow the developer to choose the best optimization, and Microsoft's C++ compiler is one the best ways to produce small, fast code.

With such power and flexibility there are potential pitfalls, of course. The chief problem with C++ is the issue of memory management, but since it is a mature language, standard solutions have been developed, for example smart pointers. There are many established standard libraries and their longevity and maturity means that they are broadly used and tested and so can be trusted to be stable.

The advantage of using a standard library is that there is a lot of documentation and examples illustrating how to use it and you have the confidence that it will deliver. For example the Standard Template Library [http://msdn.microsoft.com/en-us/library/cscc687y(v=VS.100).aspx] (STL) for C++ has existed for two decades and has been an ANSI standard since 1994.

Finally, C++ has been the language of choice for Windows development for two decades, this means that many companies have a huge investment in C++ source code, and yet the great flexibility of the language means that source code from two decades ago can be integrated into code that uses the most up-to-date Windows 7 features.

Choosing the Right Development Tools

Visual Studio has long been the development tool of choice for Windows developers. Visual C++ 2010 adds productivity tools that make writing C++ code much easier and faster. One new feature is Live Semantic Errors. Users of Microsoft Office are familiar with this productivity tool, but this is a new feature for Visual C++. The code editor displays compiler-quality syntax and semantic errors as wavy underlines as you type. When you hover the mouse over that code a tooltip appears indicating the error.

Visual C++ 2010 also contains standard C++ language features, known as the C++0x features. Some of these features make C++ code a more readable, but others will have a significant effect on the code using them. For example, one C++0x keyword is **nullptr** [http://msdn.microsoft.com/en-us/library/4ex65770(v=VS.100).aspx] , this value is used to assign a null value to a pointer, and makes your code a bit more readable. However, lambda expressions [http://msdn.microsoft.com/en-us/library/dd293608(v=VS.100).aspx] , that allow you to define anonymous function objects bring a level of functional programming to native C++ and are a powerful feature. Lambda expressions are initially quite alien to untrained eyes, but they can make code less verbose, and therefore more readable.

Choosing the Right Graphics Library

The images that you see on your monitor are put there by the computer's graphics card. There are two fundamentally different ways to tell the graphics card what color each pixel should be displayed, and crucially, how that pixel changes. These two technologies are called Graphics Device Interface [http://msdn.microsoft.com/en-us/library/dd145203(v=VS.85).aspx] (GDI) and DirectX [http://msdn.microsoft.com/en-us/library/ee416796(v=VS.85).aspx] .

GDI has been part of Windows from the very first version. In essence it is a series of C functions that allow you to indicate the color of individual pixels, or collections of pixels as line or filled shapes. GDI allows you to perform simple animation with double buffering: that is, drawing the next frame in an off-screen bitmap buffer and then rapidly copying the frame to the screen in an action known as bit-blitting after the function used to do this copy: **BitBlt** [http://msdn.microsoft.com/en-us/library/dd183370(v=VS.85).aspx] . GDI allows you to draw lines, rectangles, polygons, and ellipses both as line shapes [http://msdn.microsoft.com/en-us/library/dd145028(v=VS.85).aspx] and filled shapes [http://msdn.microsoft.com/en-us/library/dd162714(v=VS.85).aspx] . These functions are quite primitive, but are simple to use. Lines are drawn using the current selected pen [http://msdn.microsoft.com/en-us/library/dd162786(v=VS.85).aspx] and areas are filled with the currently selected brush [http://msdn.microsoft.com/en-us/library/dd183394(v=VS.85).aspx] , so you simply select a pen or brush and provide the co-ordinates and GDI changes the appropriate pixels. Similarly with text, GDI offers relatively simple text drawing functions [http://msdn.microsoft.com/en-us/library/dd144819(v=VS.85).aspx] where you select the font, provide values like the pitch and point size and GDI draws the non-anti-aliased text on screen. GDI fill routines allow developers to use solid colors, but it does not provide for gradient fills, and there is no facility to use an alpha channel to provide opacity information. (One of the few GDI that uses alpha channel information is the **AlphaBlend** [http://msdn.microsoft.com/en-us/library/dd183351(v=VS.85).aspx] function, which is an alternative to **BitBlt** for copying bitmaps.)

GDI+ [http://msdn.microsoft.com/en-us/library/ms533798(v=VS.85).aspx] , the native code graphics engine beneath the .NET Framework's Windows Forms, extends GDI to allow developers to use alpha blending and gradient fills, and to draw anti-aliased text. Although GDI+ is provided as C functions exported from a DLL, it is far more object-based than GDI and the Windows SDK for Windows 7 provides a C++ class library to encapsulate these GDI+ objects. While GDI+ is an improvement over GDI, it is still implemented with the same technology.

Perhaps the biggest issue with GDI (and therefore GDI+) is that these APIs have not been written for modern Graphics Processor Units (GPU) and consequently the GDI raster operations have to be translated into the texture and vector based operations native to modern graphics processors. Although they seem to be low level primitives, this extra translation layer means that GDI has less performance than an API that talks directly to the GPU.

DirectX technologies are written for modern GPUs, consequently the APIs reflect the capabilities of the GPUs and the programming paradigm of DirectX is as close as can be achieved to the objects used by the GPUs. To allow as many graphics cards as possible to be programmed with DirectX the underlying library will check the capabilities of the card's GPU and if the GPU does not support a feature DirectX will provide a software implementation. This means that developers write the same code regardless of the graphics cards used. A DirectX application on a fully featured graphics card will use hardware acceleration.

GPUs are designed for 3D work and consequently DirectX provides the Direct3D API [http://msdn.microsoft.com/en-us/library/ff476080(v=VS.85).aspx] which is a collection of objects accessed through COM-like interfaces.

Choosing the Right Application Framework

Using some form of application framework to help with application development is often essential since a fully-

featured Windows-based application can be a complicated beast. The aim of an application framework is to allow the developers to focus on the functionality of the application and not have to worry about the underlying plumbing code.

However, choosing an application framework involves a series of trade-offs. On the one hand a framework can provide you with ready-made implementations which you can use as-is to speed up development significantly. On the other hand, a framework can often constrain what you can do or add complexity by requiring you to extend or modify the framework in order to implement the features or behavior you require, or substantially increase the overall size of your application.

Since Hilo was developed using Visual C++ 2010 Express, the use of MFC (and ATL) which are not available with this edition was not an option. Furthermore, it was decided not to use WTL since this library is unsupported. In any case, the ethos behind the Hilo project is lightweight and flexible and so we decided to write a small common library that would be a thin wrapper above the Windows and Direct2d APIs. This provided a base on which we could write all of the Hilo applications. The Hilo Common Library provides numerous features that make writing the Hilo applications easier and faster, while maintaining the performance and size characteristics we require.

Developing For Windows 7

Windows 7 provides many innovative and compelling features and these are all available either through C functions exported from a DLL or through objects with COM interfaces. Furthermore DirectX has been enhanced in Windows 7 and is becoming the API of choice for Windows developers. DirectX and the constituent technologies of Direct2D [http://msdn.microsoft.com/en-us/library/dd370990(VS.85).aspx] , DirectX [http://msdn.microsoft.com/en-us/library/ee663274(VS.85).aspx] , DirectWrite [http://msdn.microsoft.com/en-us/library/dd368038(v=VS.85).aspx] all have COM-like interfaces. It is clear that to write a compelling modern Windows 7 application you must use a language that makes accessing COM interfaces and C library functions easy and this language is Visual C++.

Windows 7 brings enhancements to the APIs for accessing tablet features and allowing applications to provide a rich user experience. The most important feature of a Tablet PC is the touch sensitive screen. You interact with the application through finger gestures and through stylus movement. Because figure gestures are a natural movement, you expect the application to react in a natural way, so when you drag a finger across the screen, you expect the selected item to move according to your finger movement. When you use a finger gesture to “flick” an item, you expect the item to appear to be “flicked” on the screen . This requires a different type of graphics development than Windows developers have been used to. Users expect to see items in an application to behave in response to their gestures, and developers need a library that provides the facilities to provide this behavior. Enter Direct2D and the animation API [http://msdn.microsoft.com/en-us/library/dd371981(v=VS.85).aspx] .

Direct2D is built over Direct3D which itself is close to the metal of the GPU of a computer’s graphics card. The paradigm is very different to Windows GDI programming. In GDI programming the program code determines how the pixels on the screen will change and the computer’s CPU performs the calculations to change those pixels, at the same time as doing the work that the display is being animated to show. In Direct2D programming the program indicates how the display will change, and then tells the graphics card’s GPU to do the required calculations and update the display. This frees up the computer’s CPU to perform data crunching work, and results in a faster, more responsive application. Since the graphics card’s dedicated GPU performs the graphics calculations it means that more complex, high performance animations can be created.

For more details about how to program Direct2D refer to “Learn to Program for Windows in C++: Module 3 Windows Graphics” [http://msdn.microsoft.com/en-us/library/ff684175(v=VS.85).aspx] .

Componentization

Componentization means encapsulating state and code as objects. GDI is a non-object API, it has no componentization. GDI+ is object-based, but it is supplied through C functions (although the Windows SDK for Windows 7 provides a C++ class library [http://msdn.microsoft.com/en-us/library/ms533958(v=VS.85).aspx] that componentizes the API). The DirectX APIs are all object based and provided through COM-like interfaces.

COM [http://msdn.microsoft.com/en-us/library/ms680573(v=VS.85).aspx] is a platform-independent, object-oriented system for creating binary software components. COM objects can be created with a variety of programming languages and these objects can be within a single process, in other processes, even on remote computers. The COM infrastructure handles the threading, re-entrancy, and inter-process calls and this provides

location transparency, that is, the client code is the same regardless of the location of the COM object.

Perhaps the most important aspect of COM is interface programming. This is a common technique in many languages and it allows code to concentrate on the *behavior* of objects rather than the *implementation*. An interface is a collection of related methods that represents a behavior, and since a COM object can implement more than one interface it can have more than one type of behavior. All access to a COM object is through an interface pointer.

Every COM object must implement an interface called **IUnknown** [http://msdn.microsoft.com/en-us/library/ms680509(v=VS.85).aspx] and every COM interface must derive from **IUnknown** (that is, implement the methods of the **IUnknown** interface). The **IUnknown** interface has three methods, two are the **IUnknown::AddRef** [http://msdn.microsoft.com/en-us/library/ms691379(v=VS.85).aspx] and **IUnknown::Release** [http://msdn.microsoft.com/en-us/library/ms682317(v=VS.85).aspx] methods and the third is a method called **QueryInterface** [http://msdn.microsoft.com/en-us/library/ms682521(v=VS.85).aspx] . The **IUnknown::AddRef** and **IUnknown::Release** methods are used for reference counting. The rules of COM mandates that when an interface pointer is copied (including the first time it is returned when a COM object is activated) the **IUnknown::AddRef** method must be called to increment the reference count. The rules also state that when code has finished using a COM interface pointer the code must call the **IUnknown::Release** method to decrement the reference count. When the reference count reaches zero there are no more users of the interface pointer and typically the **IUnknown::Release** method deletes the COM object. Reference counting through the **IUnknown::AddRef** and **IUnknown::Release** methods allows you to define the lifetime of an object and ensure that the correct cleanup is performed when the object is no longer need. The **IUnknown::QueryInterface** method is the COM equivalent of the C++ **reinterpret_cast<>** [http://msdn.microsoft.com/en-us/library/e0w9f63b(v=VS.80).aspx] operator. The **IUnknown::QueryInterface** method takes the COM name (a 128-bit identifier) of the requested interface and if the object implements the interface the method returns a pointer to the interface.

Strictly speaking, a COM object must run in a COM apartment [http://msdn.microsoft.com/en-us/library/ms693344(v=VS.85).aspx] . Apartments are the mechanism through which COM manages threading and aspects like re-entrancy. Some of the Windows API are provided with COM-like interfaces that derive from the **IUnknown** interface but do not provide other COM features like activation or run in COM apartments. This is the case for the DirectX objects and the Windows shell objects. Interface programming is a powerful technique and you may want to use it within your own applications. You do not have to use the **IUnknown** interface to use interface programming, but since the interface is standard and well-known, it is a good place to start.

For more details about how to access COM objects with C++ refer to "Learn to Program for Windows in C++: Module 2 Using COM in Your Windows Program" [http://msdn.microsoft.com/en-us/library/ff485848(v=VS.85).aspx] .

Using C++ COM Keywords

COM objects are accessed through interface pointers, and an interface pointer is a pointer to a table of method pointers. Thus an interface is the same format as a virtual method table for a C++ class. This is fortuitous because it makes it very straightforward to implement an interface using C++ and to access an interface through an interface pointer. Typically developers create an abstract class with pure virtual methods for the interface, derive the object class from this interface class, and implement the interface methods. Visual C++ provides the **__interface** keyword and this is essentially a **typedef** for **struct** with the added behavior that all members are pure virtual.

Methods, like **IUnknown::QueryInterface**, that return an interface pointer, can return a pointer to the appropriate method pointer table by casting the object pointer to a pointer to the interface class. The C++ compiler ensures that the pointer returned is a pointer to the virtual method function pointer table. If an object implements more than one interface then you can derive the class from more than one abstract interface class.

When you create COM objects using the COM API (**CoCreateInstanceEx**) you must use the COM name, a 128-bit Globally Unique Identifier (GUID) for the object and for the name of the interface on that object. Since GUIDs are unwieldy structures the Visual C++ compiler gives you an operator and a class modifier to make your code easier to read and simpler to write. The **__declspec(uuid)** class modifier allows you to attach a GUID to a class, as a kind of tag that you can read later using the **__uuidof** operator. Neither the C++ compiler nor the COM functions will automatically read this GUID tag from an object, but your code can. However, attaching a GUID to a C++ class with **__declspec(uuid)** does not magically make your C++ class a COM object, you have to do more work

than that.

Listing 1 Definition of the IUnknown interface using Visual C++ keywords

C++

```
__declspec(uuid("00000000-0000-0000-C000-000000000046"))
IUnknown
{
   HRESULT QueryInterface(REFIID riid, void **ppvObject);
   ULONG AddRef(void);
   ULONG Release(void);
};
```

Listing 1 shows the definition of the **IUnknown** interface using the Visual C++ keywords Listing 2 shows the same code using standard C++ keywords, illustrating that an **__interface** is a **struct** with pure virtual methods.

Listing 2 Definition of the IUnknown interface using standard C++ keywords

C++

```
struct IUnknown
{
   virtual HRESULT QueryInterface(REFIID riid, void **ppvObject) = 0;
   virtual ULONG AddRef(void) = 0;
   virtual ULONG Release(void) = 0;
};
```

Using COM with C++

Hilo is not a COM server and so there is no need to implement any COM activated objects in the project. However, Hilo does access the COM-like interfaces on DirectX objects. One of the problems with using COM interfaces is ensuring that reference counting is correctly handled. As mentioned earlier, every time an interface pointer is copied **IUnknown::AddRef** must be called to increment the object reference count. Whenever the code has finished using an interface pointer **IUnknown::Release** must be called to decrement the reference count. If these reference counting rules are not followed exactly then resources will not be released early enough, and the server will suffer memory leaks.

The solution to the resource leak problem is smart pointers. A smart pointer class encapsulated the interface pointer and overrides the **–>** operator to give access to the interface pointer so that the smart pointer object can be used as if it is the interface pointer. However, the most important detail about the smart pointer class is that it has a destructor that will call **IUnknown::Release** on the interface pointer when the smart pointer goes out of scope. This means that the reference count on the object is controlled by the lifetime of the smart pointer object, so for example, you can create a smart pointer as stack variable and the lifetime of the COM object will automatically be constrained to the time that the function is running. Regardless of how the function is exited: a call to return, an exception, or just reaching the end of the function, **IUnknown::Release** will be called on the encapsulated interface pointer.

The Hilo application provides a smart pointer class called **ComPtr<>**. This class can be used on any interface pointer that derives from **IUnknown** regardless of whether the interface is on a COM object or not. Listing 3 shows the entrypoint function for the Hilo Browser application. The **BrowserApplication** class (explained in more detail in an upcoming chapter) provides the main window for the application and implements an interface called **IWindowApplication**. This is not a COM interface but it derives from the **IUnknown** interface so that reference counting can be used to manage the lifetime of instances of the **BrowserApplication** class.

The templated method, **SharedObject<>::Create**, called in Listing 3 creates an instance of the **BrowserApplication** class on the C++ heap using the **new** operator and calls **IUnknown::QueryInterface** on this object for the interface represented by the parameter.

Listing 3 The entrypoint function for the Hilo Browser application

C++

```
int APIENTRY _tWinMain(HINSTANCE, HINSTANCE, LPTSTR, int)
```

```
{
    ComPtr<IWindowApplication> mainApp;
    HRESULT hr = SharedObject<BrowserApplication>::Create(&mainApp);
    if (SUCCEEDED(hr))
    {
        hr = mainApp->RunMessageLoop();
    }

    return 0;
}
```

The **ComPtr<>** variable is declared on the stack, so when the variable goes out of scope (after the **return** statement is called) the destructor is called and this calls **IUnknown::Release** on the interface pointer, which in turn calls the **delete** operator on a pointer to the **BrowserApplication** object. This illustrates that the implementation of the **IUnknown** methods are closely coupled to the implementation of the code that creates the COM object (in this case **SharedObject<>::Create**) since the code used to delete the object must be appropriate for the code used to create it.

The **IWindowApplication** interface contains a method called **RunMessageLoop**, and the implementation of this method on the **BrowserApplication** class uses this to provide a standard Windows message loop. In Listing 3 it appears that the **RunMessageLoop** method is called on an interface pointer, however, this is not the complete story. Since **mainApp** is a **ComPtr<>** object the code in Listing 3 actually calls **operator –>** on the smart pointer class, which in turn calls the method on the encapsulated interface pointer.

Conclusion

This article discussed the rationale for choosing the technologies used to implement the Hilo applications. The next article discusses the process by which the user experience for the Hilo Browser application was designed. The design and implementation of the Hilo Common Library will be covered in the following article.

Chapter 4: Designing the Hilo User Experience

The Hilo Browser application was designed to provide a compelling, touch-enabled user experience (Ux). It is a fast, responsive, and intuitive Windows 7-based application. This article explains how the application's user experience was designed, the overall design process, the personnel who were involved, and the key design decisions that were taken.

The Design Goals

The first step of the design process was to define the overall design goals for Hilo. While these goals were not completely immutable they helped to focus decisions as the design process progressed. At key points throughout the process the design team referred back to these goals and they helped to keep the process on track. These goals are deliberately high level and broad, to allow the design team some flexibility.

Hilo's purpose is to allow the user to browse, edit and share photos. Using a touch interface on a computer with a touch-sensitive screen, the user should be able to use natural gestures to browse for photos, annotate and edit them, and then share them via an online photo sharing service.

The general design goal for Hilo was to develop one or more small applications that work ed seamlessly together, developed using Native C++ and the native Windows 7 libraries. Windows 7 was chosen as the operating system of choice because of the wealth of features that offer a modern, compelling, and integrated user experience. Native C++ was chosen as the language that offers maximum speed and efficiency and a close-to-the-metal access to Windows features. The theme of efficiency was further expounded by eschewing a large monolithic application in favor of several small efficient applications that work together to create a seamless user experience.

Bringing Together the Design Team

The design team was multi-disciplinary, made up of Ux designers, Ux managers, developers, and development project managers. The broad experience of the team allowed ideas to gestate and develop, it also allowed the ideas to be examined for viability from the perspective of the technology used to implement them and the development team skills and resources.

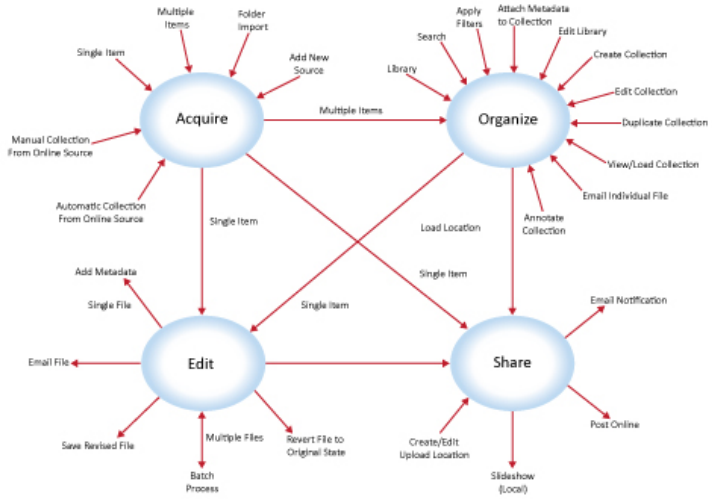
The design process was iterative, involving brainstorming sessions interspersed with periods for refinement and introspection. At key points in the process the team stepped back to examine the design according to the goals and to ensure that the team had not been distracted and deflected along a pathway that would lead away from the original goals.

Initial Brainstorming

The purpose of the early sessions was for the team to look at the application through a task-oriented lens from the user's perspective. Through these tasks the team could evaluate alternate designs, and then refine the designs that are chosen to make sure that they work across the various tasks and workflows that the user will undertake.

The first brainstorming session brought out four generic modules, collections of related features and functionality. These modules were: organize, share, edit and acquire. From the modules, a bubble diagram was drawn to identify the interconnectivity and importance of the necessary tasks. Figure 1 shows the initial bubble diagram. The four modules are connected by arrows showing the workflow and these arrows are annotated with the task name and how many items are affected by the task.

Figure 1 Initial Bubble Diagram



This was the first iteration of the design process and helped to focus the designers on the main features of the application. By focusing on the user's tasks and workflow, the actual application features started to become apparent. These features were not designed in isolation, but were designed in the context of user tasks which can have a subtle but important influence on the overall user experience.

The diagram shows that the application should be able to acquire items from different types of devices: cameras, USB thumb drives, or local drives like DVDs or hard disks; the user should also be able to acquire items from the internet. Once the items are obtained the bubble diagram shows that the user will be able to organize these items into collections, attach metadata to the collection, and add collections to the application item library. All items in the library can be shared with other users either through posting on an online photo sharing site or through a locally viewed slideshow. The final module on the bubble diagram shows that the user may edit individual items and add metadata, and then the edited file can either be saved, or shared, or the user can revert the changes.

This bubble diagram was just a first iteration, and as you will see the feature set changed as the design process progressed.

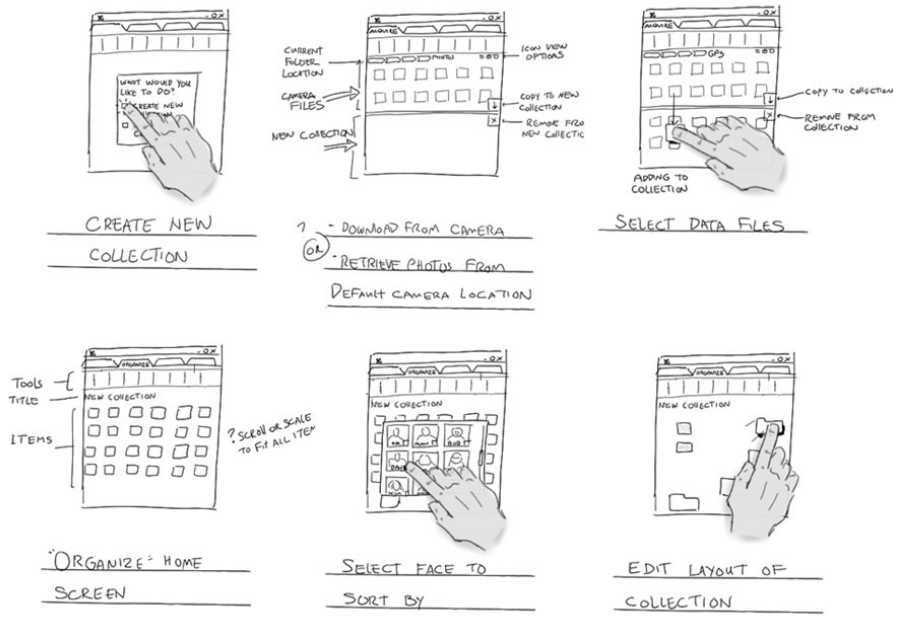
Creating Storyboards

The initial bubble diagram gave the overall range of potential features, how they relate to other features, and the workflow of the data. The next step was to use this information to create storyboards. Storyboards are very quick and rough drawings used to mock up key layouts and transitions, and to find potential issues with the design early on. This technique has been used in the film industry and is now becoming more common in user interface design.

One of the purposes of drawing everything out on a storyboard is to make sure that all the features and user interface items are represented. As the design goes through its numerous iterations, the features become more tightly integrated into the overall design and new relationships and dependencies are discovered.

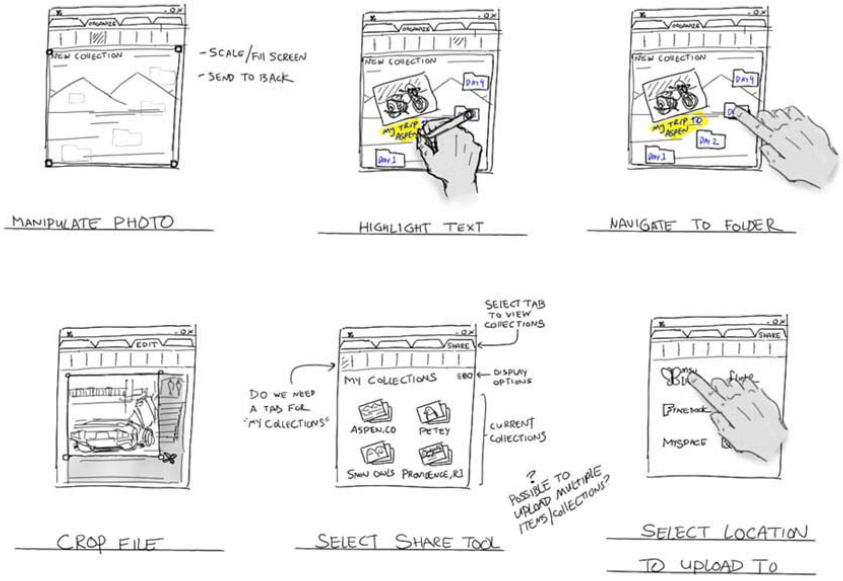
Figure 2 is a storyboard showing the actions to create a new collection of items. On this storyboard items are first acquired from a source (like a camera or a hard disk) and are displayed in the top half of the window. The user can then drag one or more items using the Touch interface of the computer screen to the collection tray in the lower half of the window. Once a collection has been created it can be organized by sorting or by manually moving items. You can see in this storyboard that the designer has thought of each option at each stage in the story and annotated the storyboard to show these options.

Figure 2 Storyboard showing the creation of a new collection of items



Another example of a storyboard is Figure 3 which shows the actions used to edit an item. This shows that the user can use the Touch interface to crop items, annotate collections and items, and organize the collections into folders by dragging items around the window.

Figure 3 Storyboard showing the actions to edit an item



Creating Mockups

The storyboards helped the design team to bring out details of how the user interacts with the application. The design team decided to produce a Silverlight prototype using Microsoft Expression Blend 3. Expression Blend is a great designer's tool since it allows designers to iterate and deliver on designer's ideas faster. By using Blend, a designer can demonstrate a vision for an application and bring an application to life, adding interactivity, animation and transitions without writing code.

The mockup allowed the team to test out the user interface design in a running environment to gauge the overall experience. The value of using a rapid prototyping environment like Blend to do this is that there is little

development cost required to get to something that the user can see and touch. This means that the design team could receive feedback from the usability testers with a short turnaround.

The Silverlight mockup of the user interface was created to test the interaction between the various modules in the application. One of the principal goals was to gauge the tactile response of the application – how it behaves dynamically – which is something that cannot be done in paper sketches. Figure 4 shows screenshots from this prototype. The mockup only included the user interface; there was no code behind the controls.

Figure 4 Silverlight prototype

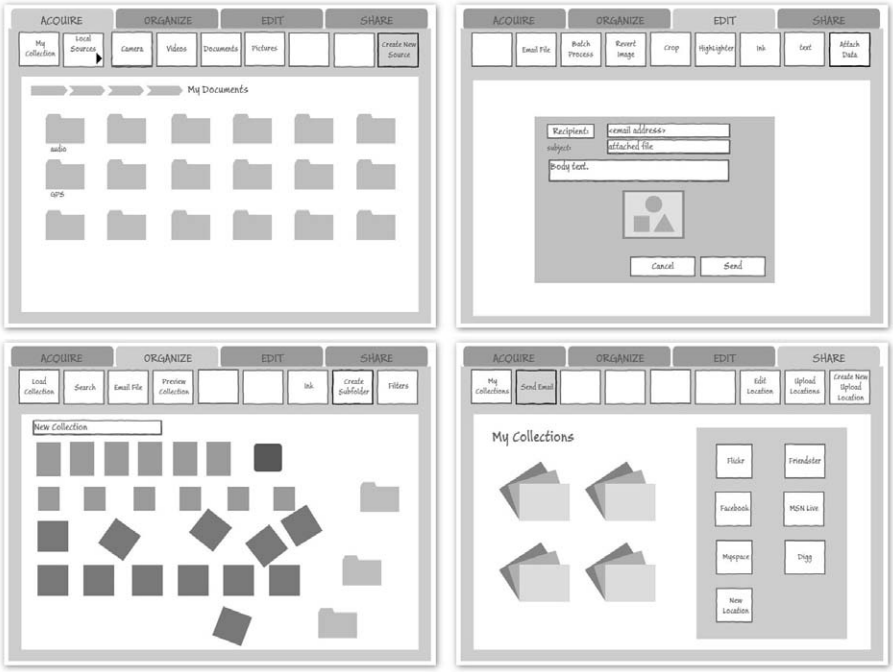


Figure 4 shows that the mockup has a tabbed layout to give access to the main modules and then a RibbonUI control to give access to the features of the selected module. The majority of the window contains the items and collections of items. Again, this mockup is just one iteration and several mockups may be created throughout a design process as new features are developed and tested.

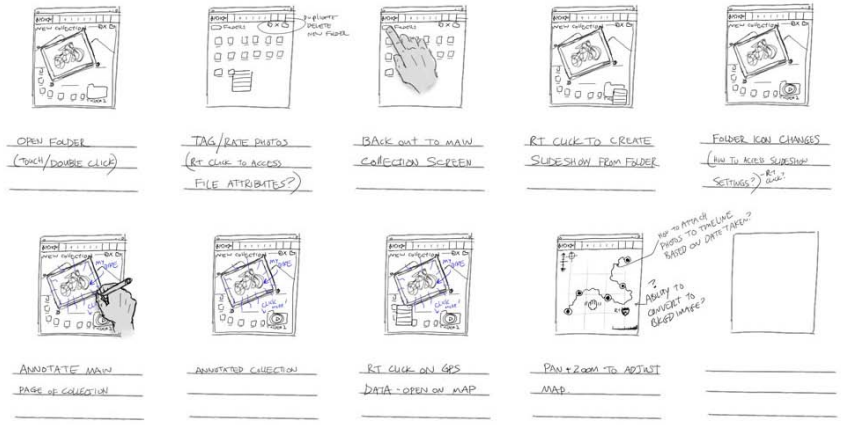
Performing Usability Tests

Now that the substantial part of the user interface had been designed, the design team commissioned usability tests to see if the design was natural and useable by ordinary users. The usability panel was chosen to be representative of a range of ages and experience. The panel was given a series of tasks to perform using the application. The usability test team monitored each member of the panel as they performed the tasks through videoing them and asking them to complete surveys. The panel was asked to express their views about how they used the application and suggestions to make the application easier and more intuitive to use.

Refining the Model

The results from the usability tests and the storyboarding helped to rationalize the design from the perspective of the user tasks and workflow. The next iteration was to redesign the model with these changes in mind. The design team decided to concentrate on the task of organization rather than treating all four modes equally and this lead to a refinement of the workflow. Figure 5 shows the storyboard drawn for this iteration with the use of a breadcrumb bar, a variation of the toolbar at the top, as well as icons representing "duplicate item," "delete item," and "new folder."

Figure 5 Further storyboard interaction showing the addition of the breadcrumb bar



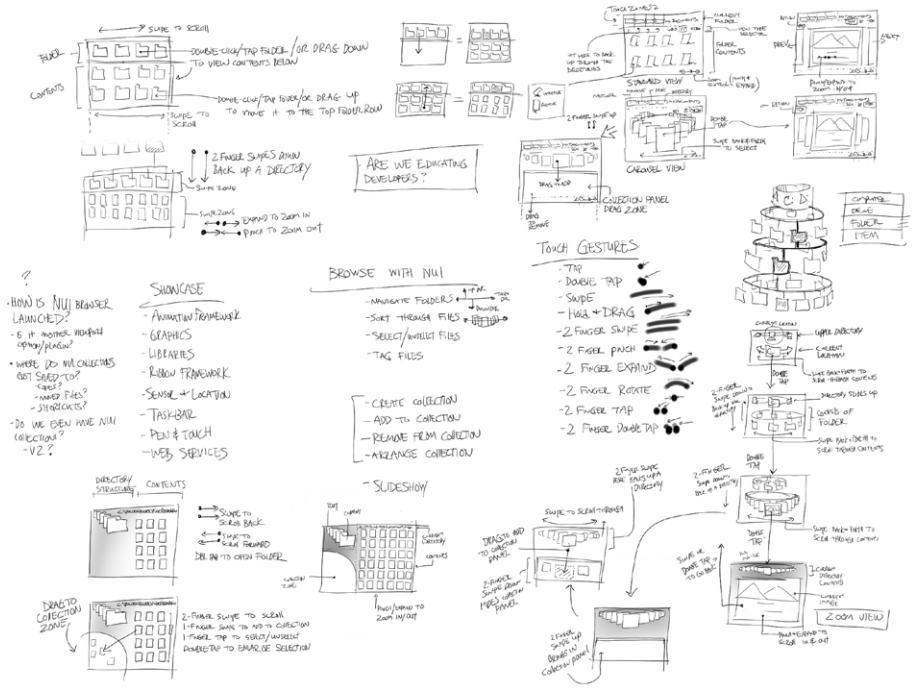
This second model showed a much cleaner workflow, however, a few issues arose concerning redundancy of buttons and tools. Discussions over these issues led to the variation of a ribbon-based user interface (UI).

Applying Introspection

At this point the design team took a step back to get a better overall view of the project. The storyboards showed that the design had progressed into quite a complex application and that there were so many features that it would increase time dramatically to get to the application into production. Since this complexity was moving the project away from the primary design goal, that is, to create a less complex application that would excite developers, the design team decided that some simplification was necessary.

Using the lessons learned in the previous storyboards and mockups, two brainstorming sessions were scheduled on consecutive days to approve a new design. For the first session the design team decided to focus solely on the browsing experience, that is, collecting items to be edited, uploaded, and so on. Four different browser designs were sketched up for discussion in the first meeting and one design is shown in Figure 6.

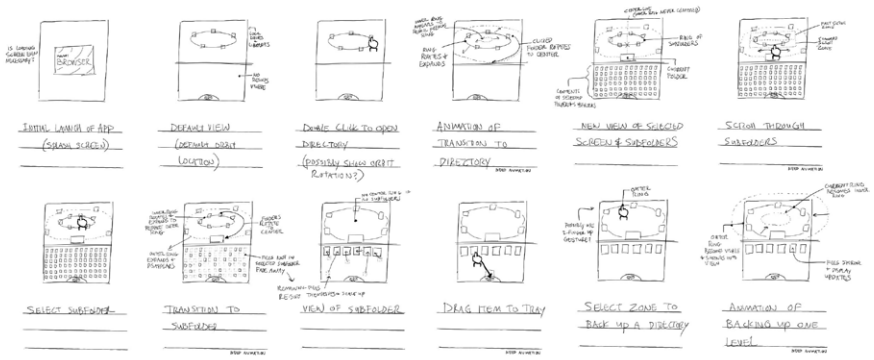
Figure 6 Revised design for the browser



As a group, the design team did some storyboarding on the whiteboard and worked through some interaction examples. After discussing the pros and cons of each design, it was decided that the hierarchical carousel (right hand side, Figure 6) had the most potential. This concept was discussed and defined further, and many helpful suggestions were provided for the next iteration, such as switching from a stacking carousel into an orbital display model. After this meeting, numerous mockups were created to be shown the following day.

During the follow up meeting the design team worked through some more variations of the design, and ultimately decided on a dual orbital design with a results pane below. Items that were selected through this browser would be stored in a collection to be accessed through the other Hilo applications later on. The next step was to storyboard an example of a browsing experience to further refine the design, as shown in Figure 7.

Figure 7 Orbital browser



The orbital display represents the file structure as it is viewed in Windows Explorer. Folders in the inner orbit are selected, causing that orbit to expand to take the place of the outer ring, which disappears. The selected folder is rotated to the bottom center of the outer orbit, and any subfolders of the selected folder appear in a new orbit in the center. The bottom pane displays the non-folder contents of the active directory, and updates anytime the current location changes. The storyboard shows the animation that occurs as the user interacts with the interface.

Continuing the Refinement

At this point the main design features have emerged. The design team created some mockups to figure out the layouts of the icons, hit and swipe zones for Touch interaction, as well as other important UI elements. Figure 8 is an example of the orbital model design, and Figure 9 has an overlay showing the touch zones.

Figure 8 Mockup of the orbital model

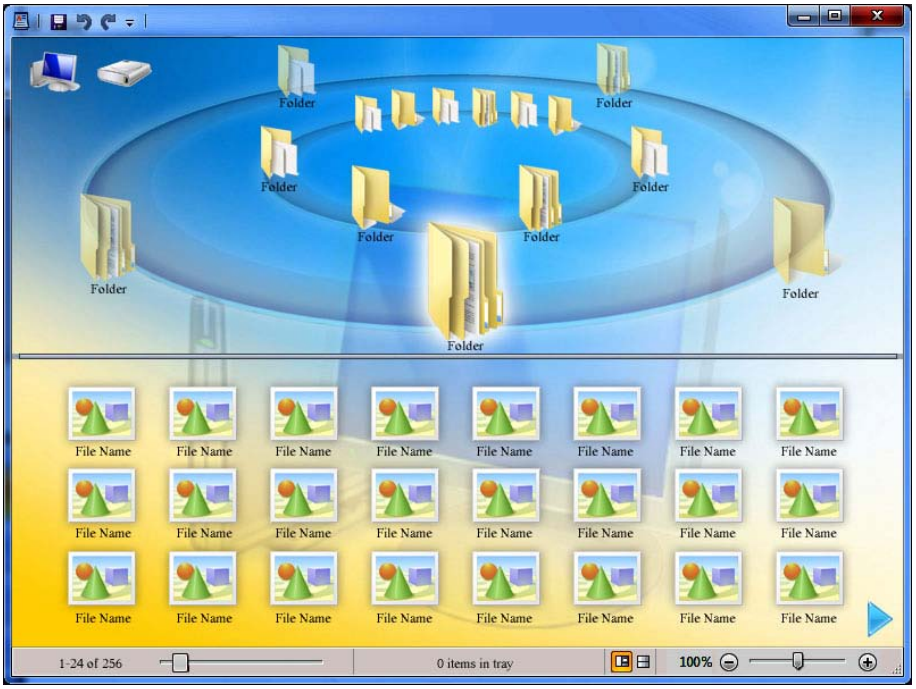
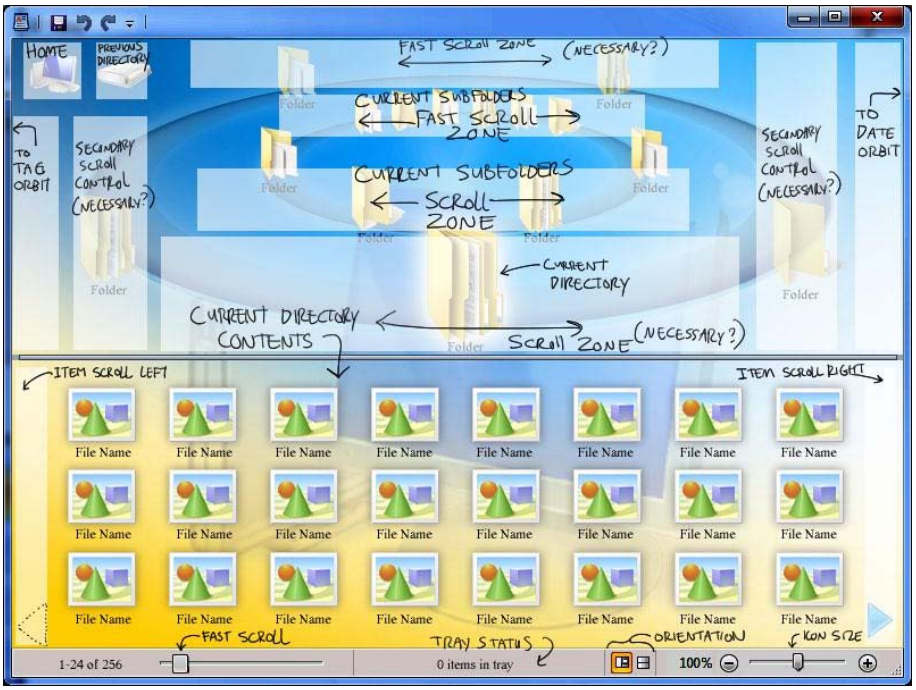


Figure 9 Mockup with an overlay of the touch zones

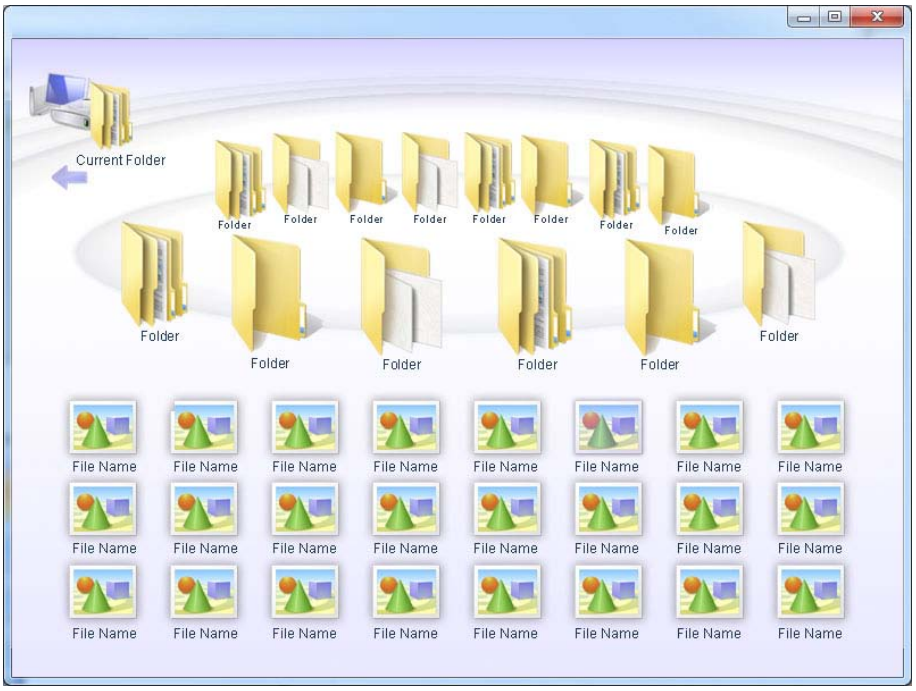


The touch and swipe zones highlighted in Figure 9 show that the user can spin the orbits with a finger or mouse gesture to bring another folder to the front, and then select this folder by touching it. In this version of the model in the upper left are icons for the Home folder and for the previously selected folder. If there are more items in the current folder than can be shown in the lower pane the user can use a finger drag gesture to bring more items into view.

These mockups led to a discussion with one of the user researchers on the team on how to best maximize the usable space for the aspect ratio and the most commonly used functions. Another storyboard was created to flesh out this revised design and further mockups were created.

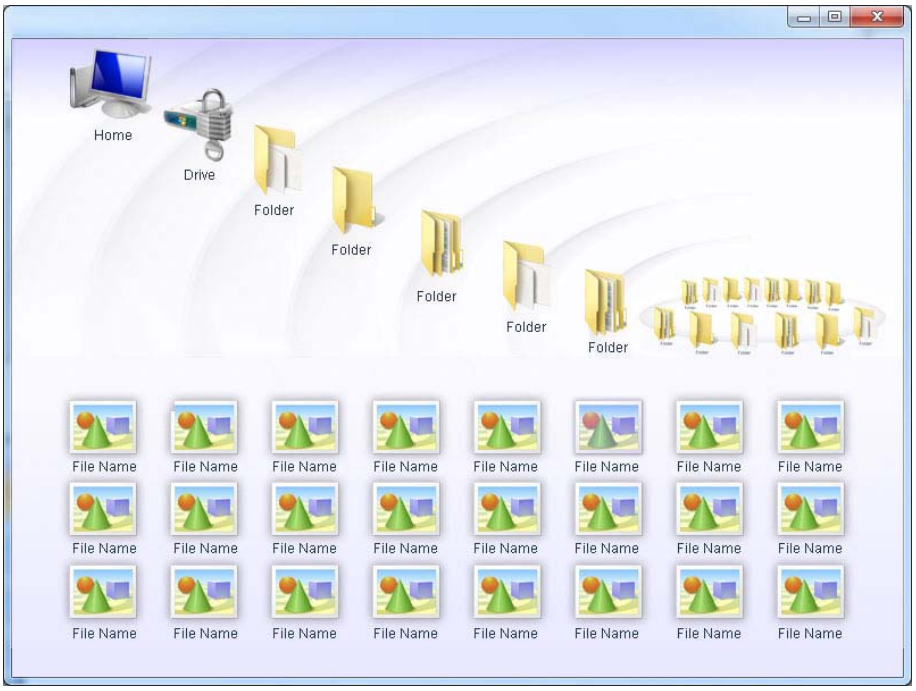
Figure 10 shows the revised orbital model. The model shows two orbits as before, the orbit for the current folder and the orbit for the current folder's subfolder. The refinement in this model is that the current folder is docked on the upper left and although the orbit for the current folder is present, it is only shown at the top. This still gives the impression of the current folder being in an orbit, but by leaving out the redundant lower half of the orbit there is more space for the inner orbit, which also allows for larger icons and more room for the thumbnail pane. Additional prototyping and testing led to the addition of a back arrow for quick navigation to the previous directory. While monitoring actual usage scenarios it was found to be used more often than the history stack.

Figure 10 Orbital model refined



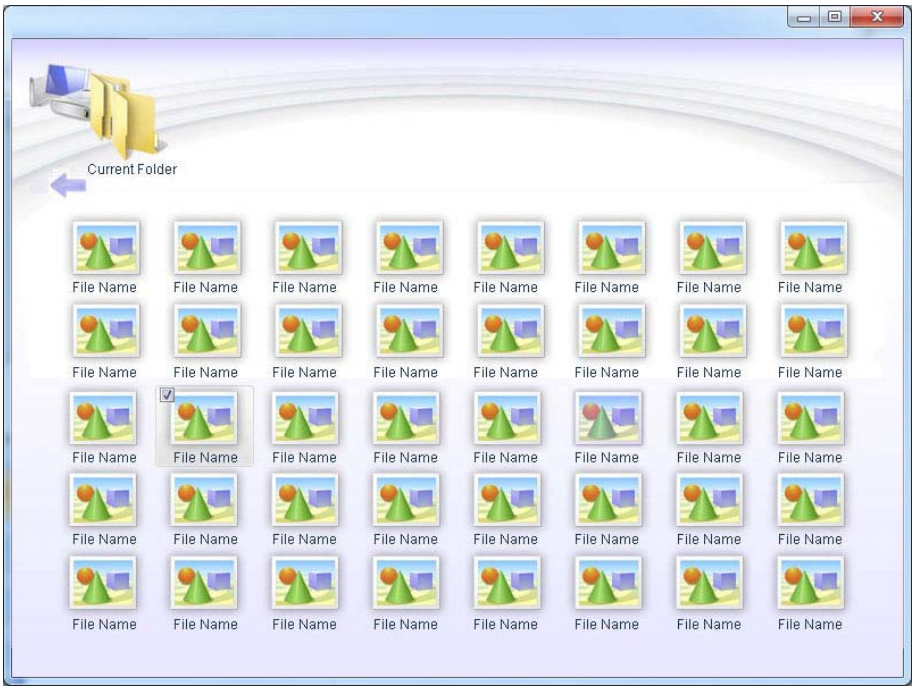
As the user drills down into the folder structure the bread crumb trail of folders is stacked up and docked at the upper left. By clicking on the stack the user can expand the bread crumb trail as shown in Figure 11, where each folder is shown as being in an orbit. The user can select any folder in the bread crumb trail to navigate to that folder.

Figure 11 Expanded bread crumb trail



If a user drills down to a folder without subfolders then the only orbit that can be show is the current folder. Figure 12 shows this situation, and shows that additional screen real estate is freed up and the thumbnail display panel in the lower half of the screen maximizes upwards to fill the available space.

Figure 12 Collapsed bread crumb trail with no subfolder orbit present



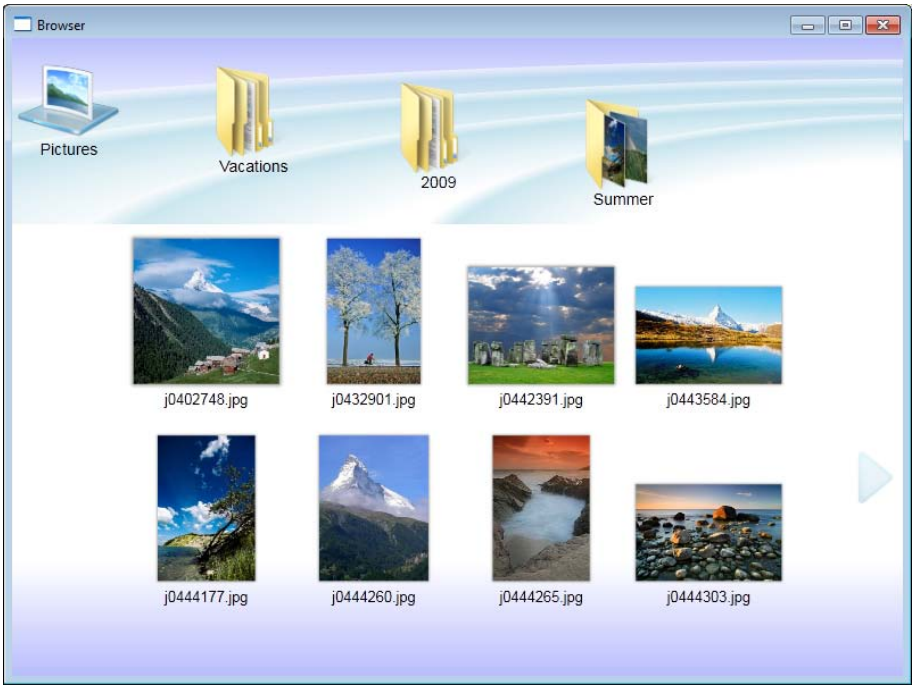
At this point the bulk of the design work had been completed.

Producing The Final Design

The final design is shown in Figure 13. This design is complete and shows where the user can interact with the UI and the effects of those interactions. This means that the design can now be handed over to the C++ development team.

This design presents a hierarchy of folders in a much cleaner way than previously seen in Windows Explorer. The designers have concentrated on the features that users focus on the most, the contents of the current selected folder, while still giving access to other folders by expanding the bread crumb trail.

Figure 13 The final orbital model



The iterative design process of brainstorming, storyboards, and mockups allowed the design team to test various scenarios and to optimize the user experience through efficient use of screen real estate and utilizing the natural

mouse and finger gestures offered by the Windows 7 Touch interface.

Conclusion

This article discussed the process by which the user experience for the Hilo Browser application was designed. The user interface for the Hilo Browser evolved through an iterative process that involved brainstorming, storyboarding, mockups, usability studies, and incremental refinement. The next article discusses the design and implementation of the Hilo Common Library, which is used as the foundation for the development of the Hilo applications.

Chapter 5: The Hilo Common Library

Hilo is made up of a number of Windows 7 applications and each has a main window and one or more child windows. Hilo implements a lightweight object orientated library to help to create and manage these windows and to handle messages sent to them. In this article we will introduce the Hilo Common Library so that future articles can focus on the features of the applications themselves and the Windows 7 features used to implement them.

Introducing the Hilo Common Library

The Hilo Common Library contains classes that are common to all the projects in the Hilo solution. These classes provide the basic infrastructure to access and provide reference-counted objects, to create windows, and to handle Windows messages.

Reference Counting

Hilo provides a template structure called **ComPtr<>** to handle reference counts. In spite of the name, this smart pointer class is not exclusive for COM interface pointers. It will work with any interface derived from **IUnknown** [http://msdn.microsoft.com/en-us/library/ms680509(VS.85).aspx] regardless of whether the object is activated by COM or runs in a COM apartment. The class implements a destructor that decrements the encapsulated interface pointer. This means that you do not need to worry about calling the **IUnknown::Release** [http://msdn.microsoft.com/en-us/library/ms682317(v=VS.85).aspx] method, the **ComPtr<>** template will do it for you when it is needed.

Many of the Hilo classes implement an interface derived from **IUnknown**. To create objects Hilo provides a template class called **SharedObject<>**. This is a mixin class: the class derives from the type provided as the template parameter so the mixin class has access to the public and protected members of the template parameter class. The **SharedObject<>** class provides static **Create** methods that create an instance of the class on the C++ heap. The **Create** methods query for an interface called **IInitializable** on the object and if it implements this interface the **Create** method calls the **Initialize** method on this interface. Code in **Initialize** is called after the object has been constructed but before it is first used. Listing 1 shows an example of the use of the **SharedObject<>** class.

The **SharedObject<>** implements the **IUnknown::AddRef** and **IUnknown::Release** methods to increment and decrement the reference count on the object. If the reference count reaches a value of zero then the **IUnknown::Release** method calls the **delete** operator to destroy the object.

The final method of the **IUnknown** method is **QueryInterface** [http://msdn.microsoft.com/en-us/library/ms682521(v=VS.85).aspx] . This method is called to request a specific interface on an object and it requires that the base class implement a helper function called **QueryInterfaceHelper** to check if the class implements the specified interface. If so, return an appropriate pointer.

Designing the Common Library

The lifetime of a Windows application is determined by the lifetime of the entrypoint function, **WinMain** [http://msdn.microsoft.com/en-us/library/ms633559(VS.85).aspx] . A Windows application is represented on screen by a window showing a frame and adornments like the close and minimize buttons, and typically closing the main window will cause the **WinMain** function to return and close the process. The process and the windows it shows are all Windows objects.

A C++ application provides C++ objects for the process and its windows. The lifetimes of the C++ objects are determined by C++ concepts like when the **new** and **delete** operators are called (for heap-based objects) and scope of the stack frame (for stack-based objects). With the Hilo Common Library a C++ object is used to provide message handlers for all the windows and this means that the lifetime of the C++ object must be longer than the lifetime of the window.

The entrypoint function for the Hilo Browser is shown in Listing 1. The **RunMessageLoop** method provides a

standard message pump that is a **while** loop that gets messages from Windows and dispatches them to the appropriate message handler function in the process. This loop executes until a **WM_QUIT** [http://msdn.microsoft.com/en-us/library/ms632641(VS.85).aspx] message is received, at which point the **RunMessageLoop** method returns and the **_tWinMain** function completes, finishing the process.

Listing 1 Entrypoint function for Hilo Browser

C++

```
int APIENTRY _tWinMain(HINSTANCE, HINSTANCE, LPTSTR, int)
{
    ComPtr<IWindowApplication> mainApp;
    HRESULT hr = SharedObject<BrowserApplication>::Create(&mainApp);
    if (SUCCEEDED(hr))
    {
        hr = mainApp->RunMessageLoop();
    }
    return 0;
}
```

In this example, the **BrowserApplication** object is created on the C++ heap by the call to **SharedObject<>::Create**, and the **ComPtr<>** smart pointer object ensures the object is deleted when the **mainApp** variable goes out of scope. The **BrowserApplication** object provides the code to handle messages sent to the main window, and to create the objects that handle the messages for its child windows.

Designing the Windows Classes

The **Window** class implements **IWindow** and is a thin wrapper over the Windows API. This class encapsulates a **HWND** value and is used to give simple access to basic Windows API windowing functions [http://msdn.microsoft.com/en-us/library/ff468919(v=VS.85).aspx] , for example to access the title, position and size of the window. Classes that provide message handlers implement the **IWindowMessageHandler** interface. The **IWindow** interface provides the methods shown in Listing 2 to give access to the object that handles the messages for a window.

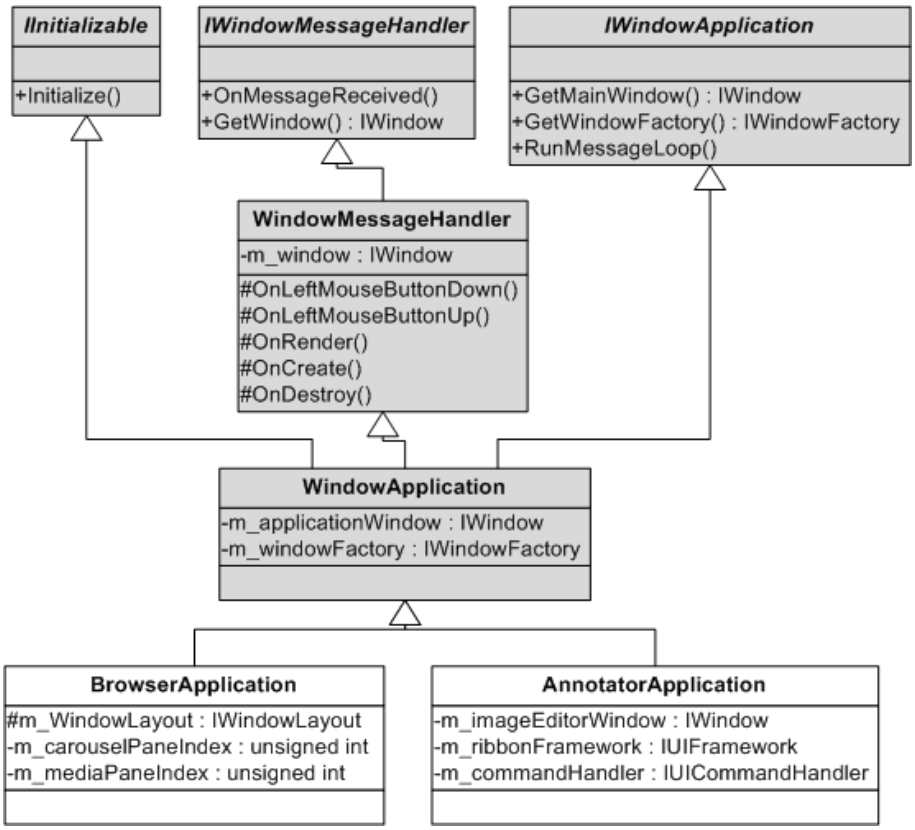
Listing 2 Methods for the Windows Message Handler

C++

```
HRESULT __stdcall GetMessageHandler(__out IWindowMessageHandler** messageHandler);
HRESULT __stdcall SetMessageHandler(__in IWindowMessageHandler* messageHandler);
```

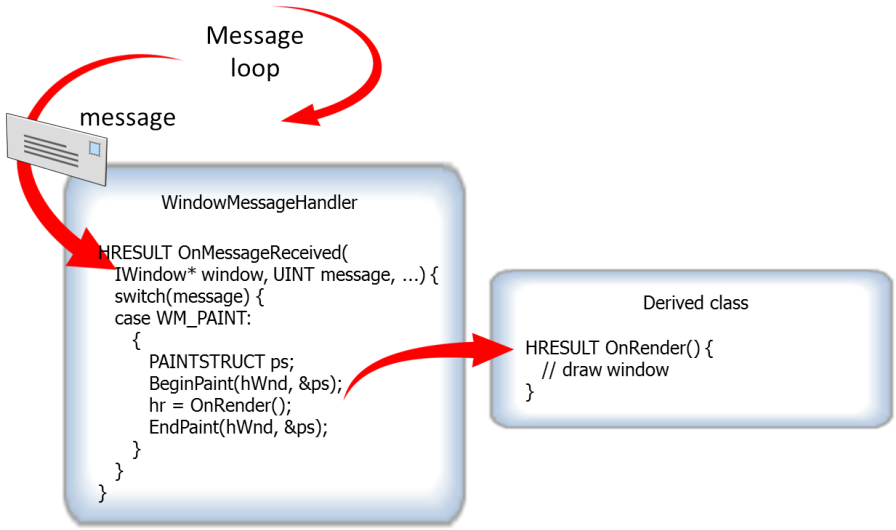
Figure 1 shows the Universal Modelling Language (UML) class diagram for the main classes in the Hilo Common Library. The **BrowserApplication** class is specific to the Hilo Browser project and the **AnnotatorApplication** class is specific to the Annotator project, the rest of the classes and interfaces are shared between all the Hilo projects and are provided through a static library project called **Common**. Figure 1 shows a selection of the attributes and operations of these classes.

Figure 1 UML diagram for the basic windowing classes, the shaded classes are provided by the Common Library



The **WindowMessageHandler** class implements a method called **IWindowMessageHandler::OnMessageReceived** to handle Windows messages sent to a window. The class does this by providing generic handling and accesses additional code by calling virtual member functions polymorphically (Figure 1 1**WM_PAINT** [http://msdn.microsoft.com/en-us/library/dd145213(VS.85).aspx] message is sent to a window the default handler code in the **OnMessageReceived** method performs standard handling of this method by calling the Windows API functions **BeginPaint** [http://msdn.microsoft.com/en-us/library/dd183362(v=VS.85).aspx] to prepare the window for painting and **EndPaint** [http://msdn.microsoft.com/en-us/library/dd162598(v=VS.85).aspx] to perform clean up. In between these function calls, the **OnMessageReceived** method calls the virtual method **OnRender** to call additional rendering code provided by the derived class. This is illustrated in Figure 2. Using virtual methods like this means that the **WindowMessageHandler** class provides default message handling for common messages and derived classes can provide additional handling.

Figure 2 Message handling in Hilo



The **WindowApplication** class implements the message loop and gives access to two objects: the main

application window and a window factory object. The window factory (implemented by the **WindowFactory** class which will be covered in a moment) creates windows through calls to the Windows API function **CreateWindow** [http://msdn.microsoft.com/en-us/library/ms632679(VS.85).aspx] . The message handling for the main application is very basic: in effect it merely ensures that a **WM_DESTROY** [http://msdn.microsoft.com/en-us/library/ms632620(VS.85).aspx] message sent to the main window will supply a **WM_QUIT** message to the message pump, which will close the process as described above.

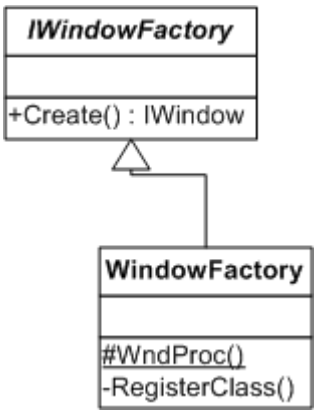
The class for the main window for each of the Hilo processes (Browser and Annotator) derives from **WindowApplication** and adds additional functionality for the main window. The **BrowserApplication** has two child windows that fill the client area of the main window and the layout of these windows is managed by an object that implements the **IWindowLayout** interface. This layout object holds **IWindow** interface pointers to both of these child windows, and the **BrowserApplication** object holds an index for each of these window so that it can access the window from the layout object. The **AnnotatorApplication** object has a window to edit images and it uses a Windows Ribbon [http://msdn.microsoft.com/en-us/library/dd371191(v=VS.85).aspx] to allow the user to access commands .

The child windows of the Hilo applications (to implement the Browser carousel and media pane, and the annotator editor) are implemented by classes derived from **WindowMessageHandler** but these are not shown in Figure 1.

Creating Windows

Typically a Windows application creates windows through a call to the **CreateWindow** function and provides the name of a registered Windows class. The Windows class is a structure that contains a pointer to a function that handles the Windows messages destined for the newly created window. The code to do this in the Hilo Common Library is a C++ class called **WindowFactory.** An instance of the **WindowFactory** class is created in the **WindowApplication::Initialize** method in order to create the application's main window.

Figure 3 UML for the WindowFactory class



The **WindowFactory** class is very simple, as can be seen by the UML in Figure 3. The **IWindowFactory::Create** method creates a window of a specified size and location, and with a specified message handler object. If the window is a child of another window the **Create** method can be passed the **IWindow** pointer to that window. The first action of the **Create** method is to call the private method **RegisterClass** which registers the static method **WindowFactory::WndProc** as the Windows procedure. This means that all windows created by the window factory object are handled by the same Windows procedure, but as you will see in a moment, the **WndProc** function forwards messages to the windows handler object for the window.

Once the **Windows** class has been registered the **IWindowFactory::Create** method creates a new instance of the **Window** object which will encapsulate the **HWND** of the new window once it is created. The **Window** object is then initialized with the message handler object through a call to **IWindow::SetMessageHandler**. Finally, the **Create** method calls the Windows API function, **CreateWindow**, to create the new window. The **CreateWindow** method is passed the **Window** object as private data through the **lpParam** parameter, and this means that this object is made available to the windows procedure. Note that at this point the encapsulated **HWND** in the **Window** object has not been assigned.

Listing 3 The WindowFactory Windows procedure

C++

```
LRESULT CALLBACK WindowFactory::WndProc(
   HWND hWnd, unsigned int message, WPARAM wParam, LPARAM lParam)
{
    LRESULT result = 0;
    ComPtr<IWindow> window;
    ComPtr<IWindowMessageHandler> handler;

    if (message == WM_NCCREATE)
    {
        LPCREATESTRUCT pcs = (LPCREATESTRUCT)lParam;
        window = reinterpret_cast<IWindow*>(pcs->lpCreateParams);
        window->SetWindowHandle(hWnd);

        ::SetWindowLongPtrW(hWnd, GWLP_USERDATA, PtrToUlong(window.GetInterface()));
    }
    else
    {
        IWindow* windowPtr = reinterpret_cast<IWindow*>(
            ::GetWindowLongPtrW(hWnd, GWLP_USERDATA));

        if (windowPtr)
        {
            window = dynamic_cast<IWindow*>(windowPtr);
        }
    }

    if (window)
    {
        window->GetMessageHandler(&handler);
    }

    if (!window || !handler || FAILED(
         handler->OnMessageReceived(window, message, wParam, lParam, &result)))
    {
        result = ::DefWindowProc(hWnd, message, wParam, lParam);
    }

    return result;
}
```

Listing 3 shows the **WndProc** function that is called for all messages sent to all windows in the Hilo applications. This method has two purposes. When the window is first created, the method will be passed the **WM_NCCREATE** [http://msdn.microsoft.com/en-us/library/ms632635(VS.85).aspx] message, and the **lpParam** parameter will represent the private data that was passed as the last parameter of the **CreateWindow** function, that is, the pointer to the IWindow interface pointer of the newly created window. The function extracts the **IWindow** interface pointer from the **lpParam** parameter and initializes the **Window** object with the **HWND** of the newly created window by calling the **IWindow::SetWindowHandle** method. Next, the code attaches the **IWindow** interface pointer to the window as user data by calling the Windows API function **SetWindowLongPtr** [http://msdn.microsoft.com/en-us/library/ms644898(VS.85).aspx] . In these actions the method has attached the C++ **Window** object (and hence also the message handler object) to the window represented by the **hWnd** parameter.

The second purpose of the **WndProc** function is to forward all Windows messages to the message handler object for the window. After the first call to the **WndProc** function, subsequent calls to the function can retrieve the user data from the **HWND** with a call to the **GetWindowLongPtr** [http://msdn.microsoft.com/en-us/library/ms633585(v=VS.85).aspx] Windows API and cast to a pointer to the **IWindow** interface. From this interface pointer the method can access the message handler object and forwards all messages to the **OnMessageReceived** method on the message handler object.

The message handler object implements the **IWindowMessageHandler** interface. When a window is destroyed it

is sent the **WM_DESTROY** message and the **IWindowMessageHandler::OnMessageReceived** method does two things. First it calls the virtual method **OnDestroy** and the default implementation in **WindowApplication** calls the **PostQuitMessage** [http://msdn.microsoft.com/en-us/library/ms644945(VS.85).aspx] Windows API function that posts the **WM_QUIT** [http://msdn.microsoft.com/en-us/library/ms632641(VS.85).aspx] message to the application window, which breaks the message pump and closes the process. The second action of the **WM_DESTROY** handler in the **OnMessageReceived** method calls **IWindow::SetMessageHandler** passing a **nullptr**. This has the effect of releasing the reference on the message handler object and destroying the object.

Conclusion

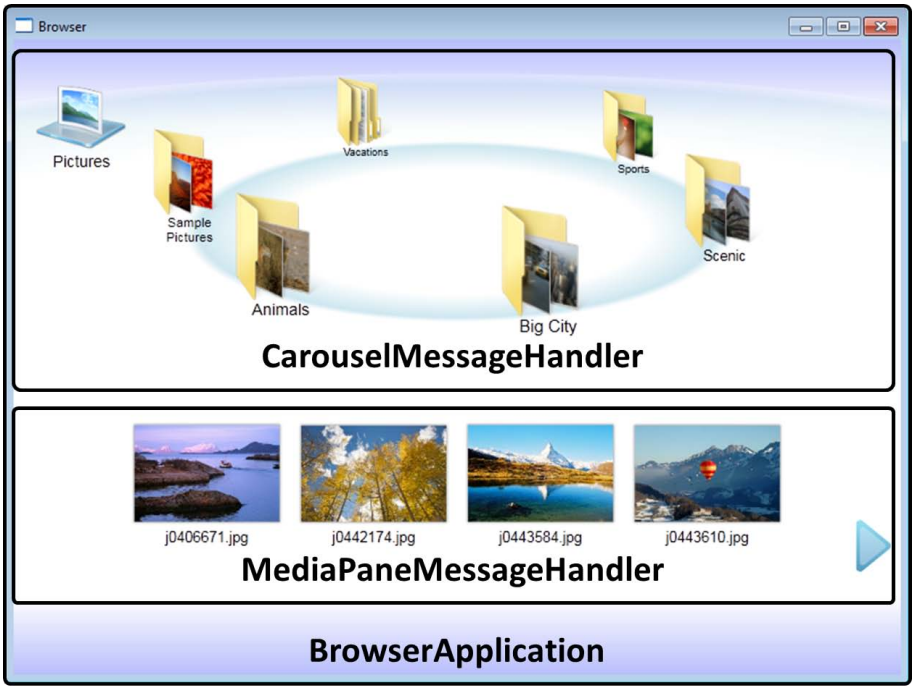
The Hilo applications use a lightweight library to provide basic Windows message handling. In this article you have seen the main classes in the library and how they interact. You have learned how these classes create and destroy windows, and how these objects relate to the C++ objects that handle Windows message. In the next article we will cover drawing on a window using the Windows 7 Direct2D API.

Putting It All Together: The Browser Application

Listing 1 shows the entrypoint function for the Browser, and shows that the main window is handled by the **BrowserApplication** class. The **SharedObject::Create** method creates an instance of this class on the C++ heap and then calls the **BrowserApplication::Initialize** method to perform post-construction initialization on the object. This method first calls the base class to create the windows factory object and uses this factory object to create the application's main window. The **Initialize** method then calls two functions **InitializeCarouselPane** and **InitializeMediaPane** to create the carousel and media pane windows as child windows of the Browser application window and Figure 4 shows the windows provided by these classes.

The main window of the Browser application contains two child windows, one for the carousel and history list and another to show the photos. The **BrowserApplication::Initialize** method creates an instance of the **WindowLayout** class to manage the size an position of these child windows. The final action of the **BrowserApplication::Initialize** method is to call the Shell API [http://msdn.microsoft.com/en-us/library/bb762136(v=VS.85).aspx] to get the Pictures library as the first folder to show in the carousel.

Figure 4 Main classes in Hilo Browser



Creating the Child Windows

The **InitializeCarouselPane** method creates a window as a child of the main Browser application window and an instance of the **CarouselPaneMessageHandler** class to handle the messages from that window. The **OnCreate**

method of the **CarouselPaneMessageHandler** class is called when the carousel window is first created and this method initializes the Direct2D resources used to draw the orbital, the folder thumbnails, and the history item stack.

The **InitializeMediaPane** method creates a window as a child of the main Browser application window and an instance of the **MediaPaneMessageHandler** class to handles the messages for that window. When the **MediaPaneMessageHandler** object is first created the **Initialize** method is called, which creates a **ThumbnailLayoutManager** object to manage the items shown in the pane. The **MediaPaneMessageHandler** class does not provide an **OnCreate** method and so only default handling is performed for the **WM_CREATE** message. The media pane also uses Direct2D and the resources are initialized immediately before they are used by the **Redraw** method (which is called as part of the handling of the **WM_PAINT** message).

Using the Layout Object

When the Hilo Browser window is resized the child windows must be resized too and this is the function of the layout object. The most important method of this class is **UpdateLayout**. This method resizes and repositions the child window to fill the client area of the main Browser window. The layout object is initialized with the height of the carousel, which is the maximum size needed to show the inner orbital, including the folder icons and the folder text. The **UpdateLayout** method sets the carousel window the height and to the width of the client area of the main browser window and it positions the carousel at the top of the client area. The **UpdateLayout** method places the media pane immediately below the carousel pane and sizes it to have the same width as the client area of the browser window and the height of the remaining space in the client area.

The media pane contains the thumbnail layout manager that calculates the positions of the navigation arrows and the images from the size of the arrows, the thumbnails, and the size of the media pane. If the media pane is large then more than one row of thumbnails can be displayed.

Painting the Windows

The base class **WindowMessageHandler** handles the **WM_PAINT** message by calling the virtual **OnRender** method, and handing the responsibility to the derived class **BrowserApplication**. This derived method accesses the layout manager to get access to the message handler objects for the carousel and media pane and calls the **IPane::Redraw** method on these handler objects, which in turn calls a method called **DrawClientArea** that draws the window. The items within the carousel and media pane windows may be animated (for example, folders spin around the carousel orbit, or thumbnails moving in the media pane) and the animation is performed by altering the position of the items in the pane between refreshes. If animation is used then the positions of the items are retrieved by the **DrawClientArea** method from the animation manager.

Handling Mouse Moves and Key Presses

Touch screen touches are treated as left mouse button clicks, in addition, the user can use the mouse or the keyboard to interact with the application. Mouse and keyboard actions are handled by appropriately named virtual methods on the message handler classes.

The **CarouselPaneMessageHandler** class handles left mouse clicks in three ways. First, if the click is in the top left corner of the pane then it is used to expand the history stack or, if the back button is clicked, navigate back in the history list. Second, the click is used to select a folder (either in the inner orbital, or if the history stack is shown, one of the items in the history stack). The third type of click, and this is the reason for the majority of the code in the **OnLeftMouseButtonDown**, **OnLeftMouseButtonUp** and **OnMouseMove** methods, is to allow the user to spin the carousel by dragging an item on the inner orbital. The user can spin the carousel by moving the mouse wheel and the **CarouselPaneMessageHandler** handles this in the **OnMouseWheel** method. If the keyboard is used, the **CarouselPaneMessageHandler** class interprets the Left and Right Arrow keys as spinning the carousel left and right, and interprets pressing the Backspace key as a signal to navigate up the history stack.

The media pane shows the images and arrows to scroll through the image list. The **MediaPaneMessageHandler** class handles the touch or mouse click events to scroll the images left and right, as well as the mouse wheel events which will also scroll the list. A left mouse click on a navigation button will scroll the list in the appropriate direction. A mouse button double-click is interpreted by the **OnLeftMouseButtonDoubleClick** method to toggle between browsing and slide view mode. In browsing mode the top half of the main window shows the carousel, but in slideshow mode the carousel panel is hidden and the currently selected image is stretched to fill as much of the Browser window as is possible while maintaining the correct aspect ratio. The user can now browser through

all the images one by one. The media pane handles key clicks in **OnKeyDown**. The Page Up and Page Down keys scroll the image list left and right (similar to clicking the left and right navigation arrows) and the Plus Sign and Minus Sign keys are used to change the size of the thumbnails.

Closing Down The Application

When the user clicks the close button a **WM_CLOSE** [http://msdn.microsoft.com/en-us/library/ms632617(VS.85).aspx] message is sent to the application window and the **WindowMessageHandler** class handles this by calling the virtual **OnClose** method. None of the Hilo classes override the **OnClose** method, and the default implementation does nothing other than calling the Windows default handing. Windows default handling of **WM_CLOSE** calls the **DestroyWindow** [http://msdn.microsoft.com/en-us/library/ms632682(v=VS.85).aspx] function which sends the **WM_DESTROY** [http://msdn.microsoft.com/en-us/library/ms632620(v=VS.85).aspx] message to the window. The **WindowMessageHandler** class handles this message by calling the **OnDestroy** virtual method. and the **BrowserApplication::OnDestroy** method calls a method called **Finalize** on the layout object. The **WindowLayout::Finalize** method iterates through all the child windows and calls a **Finalize** method on each one, which performs final cleanup on each window. Finally, the **BrowserApplication::OnDestroy** method then calls the base class implementation of **OnDestroy,** which calls the **PostQuitMessage** [http://msdn.microsoft.com/en-us/library/ms644945(VS.85).aspx] Windows API function that posts the **WM_QUIT** [http://msdn.microsoft.com/en-us/library/ms632641(VS.85).aspx] message to the application window and this message breaks the **while** loop of the message pump, which results in the process closing.

Chapter 6: Using Windows Direct2D

Modern computers have advanced graphics cards. To allow developers to use the full features of these graphics cards, Windows 7 provides the DirectX libraries, a collection of APIs for implementing high performance 2D and 3D graphics and multi-media capabilities in your applications. One of the APIs, Direct2D, as the name suggests, provides classes to draw in the x-y plane using hardware accelerated DirectX technologies. Another of the APIs, DirectWrite, provides support for high quality text rendering. This article describes how Direct2D and DirectWrite are used in the Hilo applications.

Using Direct2D

Direct2D [http://msdn.microsoft.com/en-us/library/dd370990(v=VS.85).aspx] is an object orientated library built using the Direct3D Application Programming Interface (API) that gives access to the drawing features of Graphics Processing Units (GPU) in modern graphics cards. The headers and libraries needed for C++ development for Direct2D are supplied as part of the Windows Software Development Kit (SDK) for Windows 7. The two main headers are d2d1.h and d2d1helper.h. The d2d1.h header file defines the main structures, enumerations, and interfaces in the library. The Direct2D objects are created through named methods on interfaces of other Direct2D objects, usually the D2D1 factory object created by a call to a global function called **D2D1CreateFactory** [http://msdn.microsoft.com/en-us/library/dd368034(v=VS.85).aspx] .

The d2d1helper.h header file defines a C++ namespace called **D2D1** [http://msdn.microsoft.com/en-us/library/dd368134(v=VS.85).aspx] that contains helper classes and functions to make using Direct2D easier. For example, there is a class called **D2D1::ColorF** [http://msdn.microsoft.com/en-us/library/dd370907(v=VS.85).aspx] that provides access to named color values and **D2D1::Matrix3x2F** [http://msdn.microsoft.com/en-us/library/dd372275(v=VS.85).aspx] that encapsulates the operations that can be carried out on a **D2D1_MATRIX_3X2_F** [http://msdn.microsoft.com/en-us/library/dd368132(VS.85).aspx] structure.

Hilo also uses DirectWrite through the dwrite.h header file. Similar to Direct2D, DirectWrite objects are created through a factory object which itself is created through a global method called **DWriteCreateFactory** [http://msdn.microsoft.com/en-us/library/dd368040(v=VS.85).aspx] .

Initializing the Direct2D Library

The Direct2D and DirectWrite factory objects should only be created once in the process, and Hilo does this through a helper class called **Direct2DUtility** that has static methods. These methods have static variables that are initialized the first time the method is called, and returned from subsequent calls. These local static variables are **ComPtr<>** objects and since they are static, when the entry point function, **_tWinMain**, finishes, the destructor of **ComPtr<>** is called that releases the final reference on the Direct2D object.

The Direct2D method, **D2D1CreateFactory**, has a parameter that indicates whether the Direct2D objects will be called in a multi-threaded or single threaded environment [http://msdn.microsoft.com/en-us/library/dd368104(v=VS.85).aspx] . In the former case the Direct2D library will protect any thread-sensitive operations from multi-threaded access on all the objects created through the factory object. Hilo uses a worker thread to provide asynchronous loading of images, and so the Direct2D objects are created as multi-threaded aware.

Creating the Render Target

In Hilo, when a window is first created, the **OnCreate** method is called. Each of the Hilo objects that handle messages for child windows (for example, **CarouselPaneMessageHandler** and **MediaPaneMessageHandler**) use **OnMethod** to call the **CreateDeviceIndependentResources** method that obtains interface pointers to the Direct2D and DirectWrite factory objects.

The child window message handler classes in Hilo also have a method called **CreateDeviceResources** to create

Direct2D and DirectWrite objects that are device dependent. Device dependent objects are created by the graphics card GPU and include the render target, brushes and bitmap objects. The render target object is where the drawing is carried out, the brushes and bitmaps are used to do the drawing. The lifetime of these GPU objects, and hence the wrapper Direct2D objects, is determined by the GPU. For performance reasons these objects should live as long as possible, but the GPU may indicate that these objects cannot be reused and must be recreated.

Listing 1 Hilo pattern for drawing windows

C++

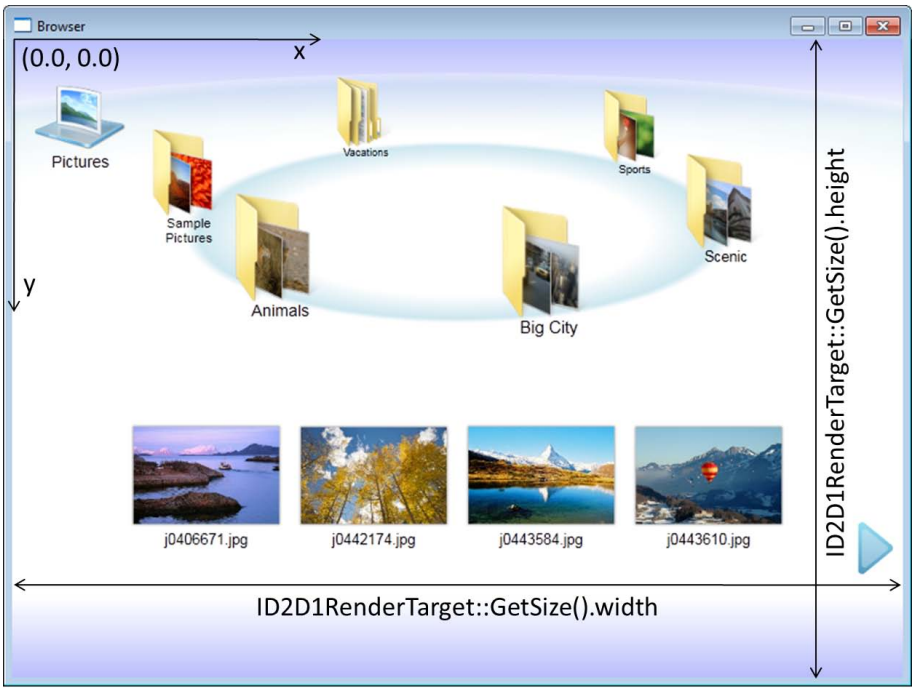
```
// Create render target and other resources if they have not been created already
HRESULT hr = CreateDeviceResources();
hr = m_renderTarget->BeginDraw();
// Do drawing here…
hr = m_renderTarget->EndDraw();

// If the GPU indicates the resources need re-creating, destroy the existing ones
if (hr == D2DERR_RECREATE_TARGET)
{
   DiscardDeviceResources();
}
```

Listing 1 shows the code pattern used in Hilo. The **CreateDeviceResources** method creates the resources if they are not already created. All drawing using Direct2D resources must be performed between calls to the **ID2D1RenderTarget::BeginDraw** [http://msdn.microsoft.com/en-us/library/dd371768(v=VS.85).aspx] and **ID2D1RenderTarget::EndDraw** [http://msdn.microsoft.com/en-us/library/dd371924(v=VS.85).aspx] methods. The **EndDraw** method returns a value indicating whether the Direct2D objects can be reused. If this method indicates that the GPU requires that the device dependent objects should be destroyed, then **DiscardDeviceResources** is called. In this case, they will be recreated by the call to the **CreateDeviceResources** method when the code in Listing 1 is called to draw the window.

Using the Coordinate System

Before you can draw on the render target you must determine where to draw. In Direct2D the drawing units are called device independent pixels [http://msdn.microsoft.com/en-us/library/dd756649(v=VS.85).aspx#what_is_a_dip?] (DIPs), a DIP is defined as 1/96 of a logical inch. Direct2D will scale the drawing units to actual pixels when the drawing occurs, and it does so by using the Windows 7 dots per inch (DPI) setting. When you draw text using DirectWrite you specify DIPs rather than points for the size of the font. DIPs are expressed as floating point numbers. By default, Direct2D coordinates have the x value increasing left to right, and the y value increasing top to bottom; the top left hand corner of a render target is x = 0.0f, y = 0.0f, this is shown in Figure 1.

Figure 1 Direct2D coordinates



You can get the size of the window in DIPs by calling the **ID2D1RenderTarget::GetSize** [http://msdn.microsoft.com/en-us/library/dd316823(v=VS.85).aspx] method. The first time you call this method it will return the size calculated from the size of the window's client area when the render target was first created. If the window changes size then Direct2D will scale its drawing to the new size of the window. This can have undesirable effects because items like text will be stretched according to how the size of the window changes. To get around this problem you can call the **ID2D1HwndRenderTarget::Resize** [http://msdn.microsoft.com/en-us/library/dd742774(v=VS.85).aspx] method and pass the size of the window in device pixel units.

Many of the items in Hilo are animated, and so their positions, size, even the opacity of the item changes with time. The change of these positions is calculated by the Windows Animation Manager and this Windows 7 component will be covered in detail in the next chapter. For now we will just say that the Hilo classes covering animation (for example, **CarouselThumbnailAnimation** is provided to animate the position of a folder on the carousel) have methods to allow the position of the animated items to be returned.

For more about Windows 7 coordinate system, see Learn to Program for Windows in C++ [http://msdn.microsoft.com/en-us/library/ff684173(v=VS.85).aspx] , in the MSDN Library.

Using Direct2D Transforms

The discussion so far about Direct2D coordinates assume that no transforms are performed. The **ID2D1RenderTarget::SetTransform** [http://msdn.microsoft.com/en-us/library/dd742857(v=VS.85).aspx] method allows you to provide a **D2D1_MATRIX_3X2_F** structure that describes the transform through a matrix. The transform is affine, which means that the matrix contains information about translation as well as rotation and scaling. To make using these matrices easier, the d2d1helper.h header file derives a class called **Matrix3x2F** from the **D2D1_MATRIX_3X2_F** structure. The **Matrix3x2F** class has static methods to return the identity matrix (no transform) or the matrix that represents a rotate, skew, scaling, or a translation. The class also has an **operator*** [http://msdn.microsoft.com/en-us/library/dd372282(v=VS.85).aspx] that you can call to combine two matrices as a single matrix.

You can call the **ID2D1RenderTarget::SetTransform** method at any time and any drawing performed afterwards through the render target will be affected by the transform. In Hilo the identity matrix is used by default, but a rotation matrix is used in one situation: drawing the navigation arrows, as shown in Listing 2.

Listing 2 Example from mediapane.cpp showing the use of a transform

C++

```
unsigned int currentPage = GetCurrentPageIndex();
unsigned int maxPages = GetMaxPagesCount();
```

```
if (currentPage > 0)
{
   // Show the left arrow
   m_renderTarget->DrawBitmap(
      m_arrowBitmap, leftArrowRectangle,
      m_leftArrowSelected || m_leftArrowClicked ? 1.0f : 0.5f);
}

if (maxPages > 0 && currentPage < maxPages - 1)
{
   // Show the right arrow by rotating the bitmap 180 deg
   m_renderTarget->SetTransform(
      D2D1::Matrix3x2F::Rotation(180.0f,
         D2D1::Point2F(
            rightArrowRectangle.left + (rightArrowRectangle.right - rightArrowRectangle.left)
/ 2.0f,
            rightArrowRectangle.top + (rightArrowRectangle.bottom - rightArrowRectangle.top) /
2.0f)));

      m_renderTarget->DrawBitmap(
         m_arrowBitmap, rightArrowRectangle,
         m_rightArrowSelected || m_rightArrowClicked ? 1.0f : 0.25f);
}
```

Handling Mouse Messages

Hilo allows the user to interact with the application through mouse clicks and the touch screen. The messages for these interactions pass the position of the mouse or finger touches using device coordinates relative to the top left corner of the client area of the window. Hilo uses these positions to determine if an item (a folder, or an image) is selected, and to do this it has to be able to convert between device pixels and DIPs. The **Direct2DUtility** class provides two static methods to do this, as shown in Listing 3. These methods use 96 DPI for the render target and calls the **ID2D1Factory::GetDesktopDpi** [http://msdn.microsoft.com/en-us/library/dd371316(v=VS.85).aspx] method to obtain the resolution for the window.

Listing 3 The methods of Direct2DUtility to convert between device coordinates and DIPs

C++

```
static POINT_2F GetMousePositionForCurrentDpi(LPARAM lParam)
{
   static POINT_2F dpi = {96, 96}; // The default DPI

   ComPtr<ID2D1Factory> factory;
   if (SUCCEEDED(GetD2DFactory(&factory)))
   {
      factory->GetDesktopDpi(&dpi.x, &dpi.y);
   }

   return D2D1::Point2F(
      static_cast<int>(static_cast<short>(LOWORD(lParam))) * 96 / dpi.x,
      static_cast<int>(static_cast<short>(HIWORD(lParam))) * 96 / dpi.y);
}

static POINT_2F GetMousePositionForCurrentDpi(float x, float y)
{
   static POINT_2F dpi = {96, 96}; // The default DPI
   ComPtr<ID2D1Factory> factory;

   f (SUCCEEDED(GetD2DFactory(&factory)))
   {
      factory->GetDesktopDpi(&dpi.x, &dpi.y);
   }
```

```
    return D2D1::Point2F(x * 96 / dpi.x, y * 96 / dpi.y);
}
```

Drawing Graphics Items

Several resources are needed in Hilo to draw items. To draw line items, filled items, or text you have to have a brush. If you already have a bitmap it can be drawn on the render target using a bitmap object. All of these items are device dependent resources and are created through method calls on a render target object. For example, all text is drawn using a font object and a brush. In Hilo the carousel draws font names using a black brush and Listing 4 shows the code to do this in **CarouselPaneMessageHandler::CreateDeviceResources**.

Listing 4 Code to create a solid color brush

C++

```
if (SUCCEEDED(hr))
{
    hr = m_renderTarget->CreateSolidColorBrush(
        D2D1::ColorF(D2D1::ColorF::Black),
        &m_fontBrush
        );
}
```

The **D2D1::ColorF** [http://msdn.microsoft.com/en-us/library/dd370907(v=VS.85).aspx] class contains static members for named colors. The color value is a 32-bit integer, so you can provide the value instead of using this class.

Creating and Using a Linear Gradient Brush

Hilo also uses gradient brushes. For example, the carousel in the Hilo Browser uses a linear gradient brush to fill the background of the carousel pane and a radial gradient brush to draw the orbitals. To create a gradient brush you need to use an additional object called a gradient stop collection. As the name suggests this object contains information about the colors that are involved and how they change.

Listing 5 Creating a linear gradient brush

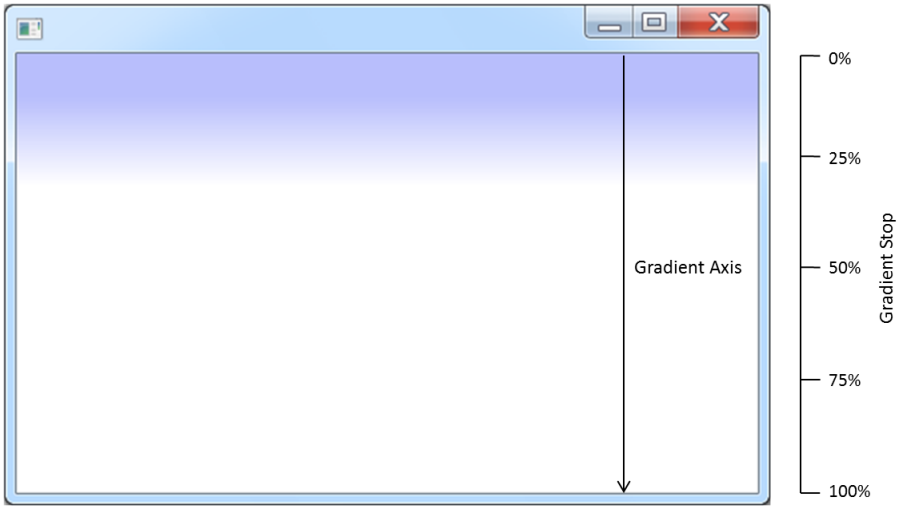
C++

```
// Create gradient brush for background
ComPtr<ID2D1GradientStopCollection> gradientStopCollection;
D2D1_GRADIENT_STOP gradientStops[2];

if (SUCCEEDED(hr))
{
    gradientStops[0].color = ColorF(BackgroundColor);
    gradientStops[0].position = 0.0f;
    gradientStops[1].color = ColorF(ColorF::White);
    gradientStops[1].position = 0.25f;
}

if (SUCCEEDED(hr))
{
    hr = m_renderTarget->CreateGradientStopCollection(
        gradientStops,
        2,
        D2D1_GAMMA_2_2,
        D2D1_EXTEND_MODE_CLAMP,
        &gradientStopCollection
        );
};

if (SUCCEEDED(hr))
```

```
{
    hr = m_renderTarget->CreateLinearGradientBrush(
        LinearGradientBrushProperties(
            Point2F(m_renderTarget->GetSize().width, 0),
            Point2F(m_renderTarget->GetSize().width, m_renderTarget->GetSize().height)),
        gradientStopCollection,
        &m_backgroundLinearGradientBrush
        );
}

gradientStopCollection = nullptr;
```

Listing 5 defines two gradient stop structures and each structure gives details of the color and the position that this color extends along the gradient axis. The important details in Listing 5 are that the gradient starts with a light purple (**BackgroundColor**) which merges with the second color, white. The gradient stop for white is at 25 percent along the gradient axis so this means that the color is solid light purple at the top, and then fades to fully white at the 25 percent position, from the 25 percent position to the 100 percent position gradient is completely white. Figure 2 shows how gradient stops are related to the changes in color. The gradient axis will be defined in a moment; the gradient stops just determine how color changes along the axis.

Figure 2 Illustrating the features of linear gradient brushes



You can have any number of stops and Direct2D will attempt to merge the colors you provide. To do this you create a stop collection object passing the information about the color stops to the **ID2D1RenderTarget::CreateGradientStopCollection** [http://msdn.microsoft.com/en-us/library/dd368113(v=VS.85).aspx] method. Finally, the **ID2D1RenderTarget::CreateLinearGradientBrush** [http://msdn.microsoft.com/en-us/library/dd371845(v=VS.85).aspx] method is called to create the brush. This method requires information about the gradient axis. The gradient axis specifies in what direction the color changes. Hilo uses the helper method from the **D2D1** namespace to initialize a **D2D1_LINEAR_GRADIENT_BRUSH_PROPERTIES** [http://msdn.microsoft.com/en-us/library/dd368128(VS.85).aspx] structure. This structure simply has two points, one describing the start and the other describing the end of the gradient axis.

The most obvious feature of the gradient axis is the angle of the axis. Listing 5 shows that the axis is a vertical line (the x position of the start point is the same as the end point), the code defines this line on the right side of the carousel window, but the same effect is achieved if the line had any x position. Figure 2 shows the gradient axis compared to the gradient stops. The gradient stops define the percentage change over the length of the gradient axis, but the **CreateLinearGradientBrush** method gives the actual length of the axis.

However, the gradient axis has some subtle features. The **CreateLinearGradientBrush** method simply creates the brush, it does not do any drawing. Listing 6 shows the code from **CarouselPaneMessageHandler::DrawClientArea** that uses the gradient brush. This code shows that the brush is used to paint an area that is exactly the same height as the gradient axis. The **ID2D1LinearGradientBrush** [http://msdn.microsoft.com/en-us/library/dd371488(v=VS.85).aspx] interface has methods to get and set the end

points of the gradient axis, so you can change these points if the area is a different size at the time it is painted than when the brush is created. In the Hilo Browser code this situation will never occur and hence the brush is used with the same property values that it was created.

Listing 6 Using a linear gradient brush

C++

```
D2D1_SIZE_F size = m_renderTarget->GetSize();
m_renderTarget->BeginDraw();
m_renderTarget->SetTransform(Matrix3x2F::Identity());
m_renderTarget->FillRectangle(RectF(
   0, 0, size.width, size.height), m_backgroundLinearGradientBrush);

// Other code

// End Direct2D rendering
hr = m_renderTarget->EndDraw();
```

Drawing Text

Direct2D allows you to draw text on the render target through the **ID2D1RenderTarget::DrawText** [http://msdn.microsoft.com/en-us/library/dd742848(v=VS.85).aspx] or **DrawTextLayout** [http://msdn.microsoft.com/en-us/library/dd371913(v=VS.85).aspx] methods. Before you can draw any text you must first create an object that holds information about the font to use and information like line spacing and text alignment. These objects are called text format objects and to create one you need to use DirectWrite. DirectWrite is similar to Direct2D in that the objects have COM-like interfaces but are created through a factory object created by a global function. In Hilo the DirectWrite factory object is created through the **Direct2DUtility::GetDWriteFactory** static method. Once you have an **IDWriteFactory** object you can create a text format object by calling the **IDWriteFactory::CreateTextFormat** [http://msdn.microsoft.com/en-us/library/dd368203(v=VS.85).aspx] method. Listing 7 shows the code used by the carousel panel in the Hilo Browser project to create the text format object. The parameters are self-explanatory, but it is worth pointing out that the size is in DIPs not points.

Listing 7 Creating a Text Format object

C++

```
// member variables:
//   ComPtr<IDWriteFactory> m_dWriteFactory;
//   ComPtr<IDWriteTextFormat> m_textFormat;

hr = m_dWriteFactory->CreateTextFormat(
   L"Arial",
   nullptr,
   DWRITE_FONT_WEIGHT_REGULAR,
   DWRITE_FONT_STYLE_NORMAL,
   DWRITE_FONT_STRETCH_NORMAL,
   12,
   L"en-us",
   &m_textFormat
   );
```

Once you have a text format object you can draw text to the render target with a call to **DrawText**. The simplest way to call this method is to pass five parameters: the string and its length, the text format object, a rectangle where to draw the text ,and a brush. The **DrawTextLayout** method is similar, but instead it is passed a text layout object that encapsulated the string, the text format object and the size of the area where to draw the text. Listing 8 shows the code from the **CarouselThumbnail::CreateTextLayout** method that creates a text layout object.

Listing 8 Creating a Text Layout object

C++

```
HRESULT hr = S_OK;

// Set the text alignment
m_renderingParameters.textFormat->SetTextAlignment(DWRITE_TEXT_ALIGNMENT_CENTER);

if (nullptr == m_textLayout)
{
   hr = m_dWriteFactory->CreateTextLayout(
      m_thumbnailInfo.title.c_str(),
      static_cast<unsigned int>(m_thumbnailInfo.title.length()),
      m_renderingParameters.textFormat,
      m_rect.right - m_rect.left,
      16,
      &m_textLayout
      );
}
```

In this code the **m_renderingParameters** member variable is set by the carousel object with a reference to the rendering target, the text format object and the brush used to draw the text. The actual text for the object is drawn in the **CarouselThumbnail::Draw** method, shown in Listing 9. The first parameter is the location to draw the text (since the format object is set to center align text, Listing 8, this point is the center of the text). The second parameter is the text layout object which contains details of the text and the font to use, the third parameter is the brush and finally there is a parameter for options.

Listing 9 Drawing text

C++

```
m_renderingParameters.renderTarget->DrawTextLayout(
   D2D1::Point2F(m_rect.left, m_rect.bottom),
   m_textLayout,
   m_renderingParameters.solidBrush,
   D2D1_DRAW_TEXT_OPTIONS_CLIP);
```

Conclusion

In this chapter you have learned how to used Direct2D to draw in a window. You learned how to initialize the Direct2D environment, create resources and draw items and text. In the next chapter you will learn how to use the Windows Animation Manager to control the positions of the items that you draw in a window.

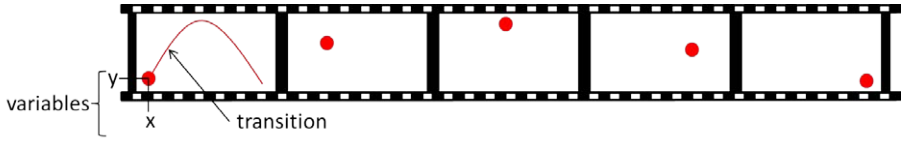
Chapter 7: Using Windows Animation Manager

Animation is the process through which an image changes over time, and if the interval between the changes is small, then to the human eye the changes appear continuous and give the impression of movement. Some animations may be simple linear changes with time, others may be more complex involving changes that accelerate or decelerate, and some animations are built up of several component movements. All of this can involve some quite complex coding and so to address this challenge Windows 7 provides a component called the Windows Animation Manager.

Using The Windows Animation Manager

Animation can be viewed as a series of frames in a storyboard. Each frame is made up of items and the changes to these items between frames makes up the animation. The changes may be made to the color, position or shape of the items, and the changes may be smooth or discrete, and if smooth, the change could be constant, or the rate of change may increase or decrease. In some cases several properties of an item change. Figure 1 shows a simple storyboard made up of five frames where a red circle changes position on each frame.

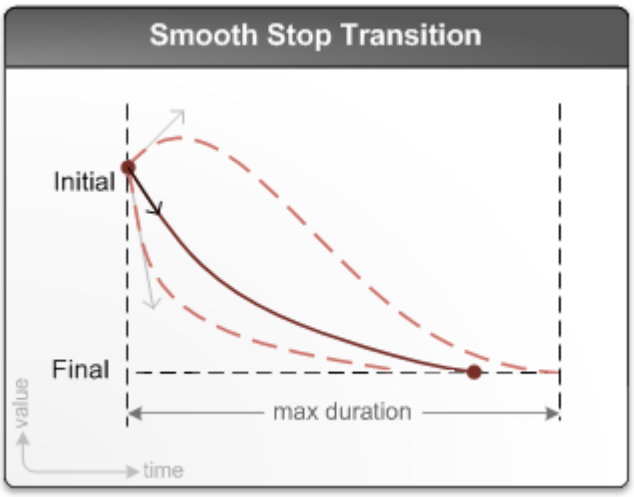
Figure 1 Storyboard showing variables and transition



In the terminology of the Windows Animation Manager, these changes are called *transitions*, and they change a floating point value called a *variable*. The animation manager does not change pixels on the screen, it just changes the variable, and you have to provide the drawing code that uses the variable. A transition determines how a variable changes over a period of time (called the *duration*) and the variable will start the transition with an initial value.

Transitions can be chained, that is, one transition describing the change of a variable is followed by another transition describing a different change to the same variable. At the hand over point between the transitions the variable will be changing in a particular way according to the first transition (called the *velocity*), and since some transitions depend on the velocity this value is made available to the next transition. Figure 2 shows a smooth stop transition and illustrates the initial and final values and the duration of the transition. Other transition types are supported by the Windows Animation Manager—for details see the Storyboard Overview [http://msdn.microsoft.com/en-us/library/dd756779(VS.85).aspx] in the Windows Animation Development Guide available on MSDN. The arrows indicate the initial velocity of the transition. Over the duration of the smooth stop transition the variable will decrease from the initial value to the final value.

Figure 2 Illustrating the properties of a smooth stop transition



The animation manager will ensure that the variable's value changes over the duration in the manner specified, and the application obtains the value of the variable during this time to render each frame. The variable clearly depends on the time since transition started, so the animation manager must have an accurate value of the time the transition starts, and the time when each frame is drawn. Windows provides a high precision timer object that can be used by the animation manager. In addition, Windows also defines several stock transitions through an object called the transition library.

Animation involves more than just changing pixels on a screen. Careful attention must be made to prevent flicker through synchronization with the monitor refresh. Direct2D does this for you, and so the combination of the Windows Animation Manager and Direct2D are the best way to provide animation in a Windows 7-based application.

Creating the Windows Animation Manager

The Windows Animation Manager is a COM object, and it must be created in a single-threaded COM apartment (STA).The thread that will use the animation manager (typically the thread that handles the messages for a window) must be initialized with a call to **CoInitialize** [http://msdn.microsoft.com/en-us/library/ms678543(VS.85).aspx] . Once an application has created an animation manager it can create a storyboard object which encapsulates one or more transitions. Listing 1 shows the code to create the main objects, these objects will be used by all animations and so typically these will be global to the application.

Listing 1 Creating the animation objects

C++

```
// Objects used by the application
IUIAnimationManager* g_animationManager = nullptr;
IUIAnimationTimer* g_animationTimer = nullptr;
IUIAnimationTransitionLibrary* g_transitionLibrary = nullptr;

CoCreateInstance(
        CLSID_UIAnimationManager,
        nullptr,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&g_animationManager)
        );

CoCreateInstance(
        CLSID_UIAnimationTimer,
        nullptr,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&g_animationTimer)
        );
CoCreateInstance(
        CLSID_UIAnimationTransitionLibrary,
```

```
        nullptr,
        CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&g_transitionLibrary)
        );
    }
```

The animation manager must be updated periodically with the time and this is the reason for the animation timer object. The transition library is a factory object that is used to provide transition objects for most of the common transitions used in animation.

Creating Animation Variables

The animation manager updates an animation variable during the duration of the animation. The change of the animation variable is determined by transition objects and so an animation variable must be associated with one or more transition objects, therefore the variable must be accessible to the code that creates the transitions. The variable has a value that will be used to draw the screen, so the variable must be accessible by the rendering code. The variable may live just for the time of a single animation, or if the animation will be repeated starting at the position it ended previously then the variable object may live for several animations. These implementation details are determined by the developer, but the important point is that a variable object may live longer than the transition it is attached to and be accessible by different methods in the application.

Listing 2 shows how to create an animation variable. The call to **IUIAnimationManager::CreateAnimationVariable** [http://msdn.microsoft.com/en-us/library/dd371699(v=VS.85).aspx] creates a variable with the specified initial value. The application can use the **IUIAnimationVariable** [http://msdn.microsoft.com/en-us/library/dd316797(v=VS.85).aspx] interface to provide maximum and minimum limits by calling the **SetUpperBound** [http://msdn.microsoft.com/en-us/library/dd317010(v=VS.85).aspx] and **SetLowerBound** [http://msdn.microsoft.com/en-us/library/dd317005(v=VS.85).aspx] methods and it can obtain the current value of the variable by calling **GetValue** [http://msdn.microsoft.com/en-us/library/dd317002(v=VS.85).aspx] .

Listing 2 Creating an animation variable

C++

```
// The variable is declared elsewhere
IUIAnimationVariable* m_variable;

if (nullptr == m_variable)
{
    hr = g_animationManager->CreateAnimationVariable(initialValue, &m_variable);
}
```

Creating and Initializing a Storyboard Object

When the application starts the animation it creates a storyboard object. The storyboard contains one or more transitions that describe the animation. The transition objects and the storyboard live just for the duration of the animation. If you want to repeat the animation then you have to create a new instance of the storyboard and transitions.

Listing 3 shows the code to create a storyboard object. This object is created through a call to the **IUIAnimationManager::CreateStoryboard** [http://msdn.microsoft.com/en-us/library/dd371703(v=VS.85).aspx] method that returns a reference to the storyboard object. When you call this method, the animation manager will hold an additional reference to the storyboard for the duration of all the transitions that are part of the storyboard. This is why the code in Listing 3 releases the reference to the storyboard object when the application has finished initializing it.

The storyboard is initialized with one or more transitions, and each transition is related to a specific variable. In Listing 3 the transition library is used to create an accelerate-decelerate transition. The duration of the transition is specified by the second parameter and the variable will have a value of **endValue** when the transition is complete. The shape of this transition is determined by the third and fourth parameters, the value of **accRatio** specifies the proportion of the duration spent initially accelerating, and **decRatio** specifies the proportion of the duration spend decelerating to the final value. This transition object is then added to the storyboard along with the

variable that it will change. Adding the transition to the storyboard means that the storyboard holds a reference to the transition object and so once the code has finished initializing the transition it releases its reference. The lifetime of the transition is now controlled by the storyboard object and the lifetime of the storyboard object is controlled by the animation manager.

Listing 3 Initializing the Animation Manager with a storyboard

C++

```
IUIAnimationStoryboard* storyboard;
IUIAnimationTransition* transition;

HRESULT hr = g_animationManager->CreateStoryboard(&storyboard);

if (SUCCEEDED(hr))
{
    hr = g_transitionLibrary->CreateAccelerateDecelerateTransition(
                duration,
                endValue,
                accRatio,
                decRatio,
                &transition);

    if (SUCCEEDED(hr))
    {
        hr = storyboard->AddTransition(m_variable, transition);
    }

    // Schedule the storyboard (see Listing 5)
}

transition->Release();
storyboard->Release();
```

The **IUIAnimationTransitionLibrary::CreateAccelerateDecelerateTransition** [http://msdn.microsoft.com/en-us/library/dd371900(v=VS.85).aspx] method creates a transition that lasts for a fixed time, the duration parameter. The transition library also allows you to create transitions whose duration depends on values determined at runtime. For example, Listing 4 shows code that uses the **IUIAnimationTransitionLibrary::CreateLinearTransitionFromSpeed** [http://msdn.microsoft.com/en-us/library/dd371925(v=VS.85).aspx] method. This method is initialized with just the initial rate of change of the variable (**speed**) and the expected final value. The duration of the transition will depend on the initial value of the variable immediately before the transition starts.

Listing 4 Creating a transition without a known duration

C++

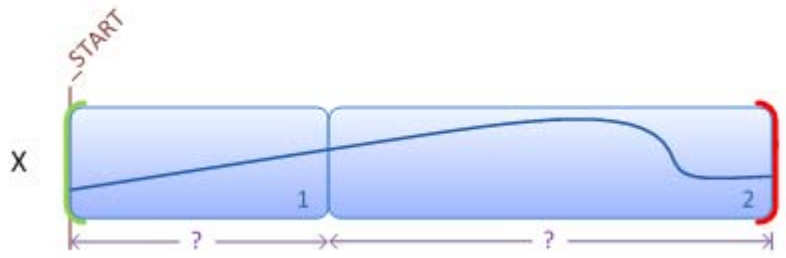
```
hr = g_animationManager->CreateAnimationVariable(initialValue, &m_variable);

if (SUCCEEDED(hr))
{
    hr = g_transitionLibrary->CreateLinearTransitionFromSpeed(
                speed,
                endValue,
                &transition);

    if (SUCCEEDED(hr))
    {
        hr = storyboard->AddTransition(m_variable, transition);
    }
}
```

The application can use the **IUIAnimationTransition** [http://msdn.microsoft.com/en-us/library/dd371887(v=VS.85).aspx] interface to set the initial value of the variable and the initial velocity (rate of change) of the variable.

Chaining Transitions

A storyboard can contain more than one variable and a variable can be added to the storyboard associated with different transitions. Transitions added to a storyboard and associated with the same variable are run serially. That is, the first transition added to the storyboard is run for its duration and then the second transition is run. This is illustrated in Figure 3 where a variable **1** is added twice to the storyboard which means that the two transitions are performed in the order that they are added to the storyboard.

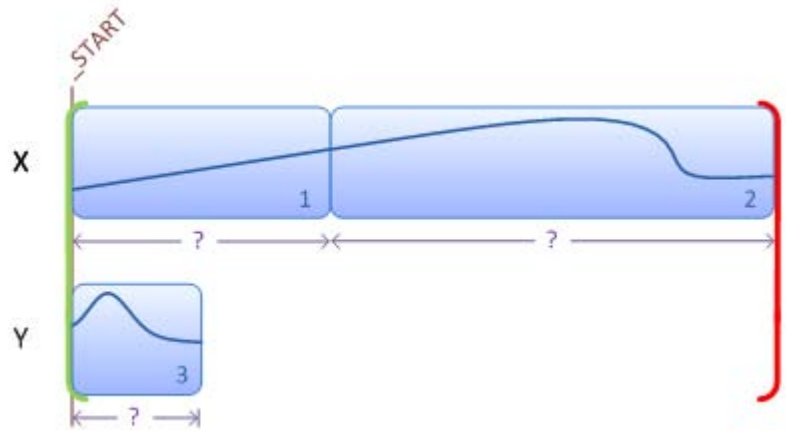
Figure 3 Chaining two transitions on the same variable



If a storyboard is initialized with transitions that are associated with different variables then when the animation starts the two variables will be updated in parallel. This is illustrated by Figure 4 where three transitions are added to the storyboard and these are associated with two1**X** and **Y**. This means that when the animation starts the first transition associated with **X** and the first transition associated with **Y** are started at the same time and so the variables **X** and **Y** are changed in parallel.

When you add a transition for another variable by using the **IUIAnimationStoryboard::AddTransition** [http://msdn.microsoft.com/en-us/library/dd371789(v=VS.85).aspx] method the two transitions will be scheduled to start at the same time. You can specify that a transition starts at a later time by creating a key frame. A key frame is essentially a defined point in time after the storyboard has started. You can create a key frame by calling the **IUIAnimationStoryboard::AddKeyframeAtOffset** [http://msdn.microsoft.com/en-us/library/dd371784(v=VS.85).aspx] method. Once you have created a key frame you can add a transition at this point by calling the **IUIAnimationStoryboard::AddTransitionAtKeyFrame** [http://msdn.microsoft.com/en-us/library/dd371795(v=VS.85).aspx] method.

Figure 4 Illustrating two variables added to the storyboard



Scheduling a Storyboard

The code in Listing 3 creates and initializes the storyboard, to start the animation you have to schedule it, as shown in Listing 5.

Listing 5 Scheduling the storyboard

C++

```
if (SUCCEEDED(hr))
{
    UI_ANIMATION_SECONDS secondsNow = static_cast<UI_ANIMATION_SECONDS>(0);
    if (SUCCEEDED(hr))
    {
        hr = g_animationTimer->GetTime(&secondsNow);
    }

    if (SUCCEEDED(hr))
    {
        hr = storyboard->Schedule(secondsNow);
    }
}
```

The **IUIAnimationStoryBoard::Schedule** [http://msdn.microsoft.com/en-us/library/dd371820(v=VS.85).aspx] method schedules the storyboard to run. If there is no other storyboard with the same variables already scheduled to run the storyboard will start immediately. Only one storyboard with the same collection of variables can run at any time, but a new storyboard can be initialized to cancel, trim, conclude, or compress conflicting storyboards.

Obtaining the Value of a Variable

At this point the storyboard is initialized and started. The storyboard will run for the duration of all the transitions, and will update the variable objects. It is now up to the application to obtain the values of the variables and use them to determine how the window is drawn. The storyboard is scheduled with the start time and so the animation manager will know at what point it currently is in each transition (and hence the value of the variables). Thus, before the application can obtain the value of a variable it must first update the animation manager with the current time. The code to do this is shown in Listing 6.

Listing 6 Code to update the animation manager

C++

```
UI_ANIMATION_SECONDS secondsNow = 0;
hr = g_animationTimer->GetTime(&secondsNow);

if (SUCCEEDED(hr))
{
    hr = g_animationManager->Update(secondsNow);
}
```

When the application calls the **IUIAnimationManager::Update** [http://msdn.microsoft.com/en-us/library/dd371755(v=VS.85).aspx] method the animation manager updates all the variables, and you can then obtain the value of a variable by calling the **IUIAnimationVariable::GetValue** [http://msdn.microsoft.com/en-us/library/dd317002(v=VS.85).aspx] method, as shown in Listing 7.

Listing 7 Retrieving the value of a variable

C++

```
double value;
m_variable->GetValue(&value);
```

The window rendering code will use the variable value to draw the new frame of the animation. The rendering code needs to know if there are more frames to be drawn in the animation and the animation manager will be able to calculate this from the current time. If there is time for more frames to be drawn then the rendering must be called again. Listing 8 shows how to determine if more frames can be drawn by obtaining the status of the animation manager.

Listing 8 Determining whether the animation has finished

C++

```
UI_ANIMATION_MANAGER_STATUS status;
hr = g_animationManager->GetStatus(&status);
if (SUCCEEDED(hr))
{
    if (status == UI_ANIMATION_MANAGER_BUSY)
    {
        InvalidateRect(hWnd, NULL, FALSE);
    }
}
```

There are two possible status codes, **UI_ANIMATION_MANAGER_IDLE** indicates that all storyboards have completed and so the rendering code need not be called again. **UI_ANIMATION_MANAGER_BUSY** indicates that there is at least one storyboard running or scheduled to run, and hence the code in Listing 8 invalidates the window to ensure that the rendering code is called again to display the next frame of the animation.

Using the Windows Animation Manager in Hilo

The basics of animation using the Windows Animation Manager have been covered in the sections above. The Hilo Browser project contains classes that provide the various animations that are used in the Browser application. Each class encapsulates one or more animation variables and provides methods to provide the initial values for these variables and create and schedule the storyboard. The main animation classes in the Browser project are described in the following table along with a description of the values that are changed as part of the animation.

Class	Description
CarouselAnimation	Rotates the folder thumbnails on the inner orbital. Called when you spin the carousel. In the animation, the folder thumbnails change size, opacity, and angular position on the orbital.
CarouselThumbnailAnimation	Moves a folder thumbnail to the history stack. Called when you select a folder. In the animation, the folder thumbnails change their x and y position and opacity.
FlyerAnimation	Fills the media pane with images. Called when a folder is first opened, so old images fly out and new images fly in. In the animation, images move along a predefined path.
MoverAnimation	Moves images in the media pane when you resize the Browser window. Images move to fill in free space as the window increases. In the animation the x and y position changes.
OrbitAnimation	Animates the size and position of the inner orbital. This Called when you select a folder and the new inner orbital comes into view, and when the history stack is expanded. In the animation, the position, the size and the opacity of the ellipse changes.
SlideAnimation	Fills the media pane with images. Called when you move to another page so new images slide into view. In the animation the x position of the images changes.

The classes have a similar form, with a two stage construction pattern where there is an **Initialize** method to create the animation variables that will live the lifetime of the instance of the class. These classes have a method to create and schedule the storyboard (typically called **Setup**), and in the process create transitions and associate

them with the animation variables. Finally, they all have one or more methods to obtain the values of the animation variables during the animation.

Instances of these classes are created by a separate class, **AnimationFactoryImpl**, which contains methods that wrap calls to **SharedObject<>::Create** to create instances of the appropriate class on the C++ heap.

The Hilo Browser animates three types of objects: the carousel (the folder thumbnails and the inner orbital), the folder thumbnails on the history stack, and the photo thumbnails in the media pane.

Initializing the Windows Animation Manager Objects

The Hilo Common Library provides a class called **AnimationUtility** with static methods that give access to the main animation objects and provides standard code used in animations. The **AnimationUtility** class has static members for the animation manager, animation timer, and transition library. A private method, **Initialize**, creates all three objects if they are not already created. The accessor methods return an interface pointer to these objects, so it means that the first time one accessor method is called all three objects are created. The **AnimationUtility** class also has a method to check the state of the animation manager and a method to schedule a storyboard.

The Hilo animation classes all have an **Initialize** method that accesses the animation manager to create the animation variables used by the animation. The lifetime of the animation variables is the lifetime of the animation object. The handler objects for the Browser carousel pane and the media pane create these animation objects using an animation factory object, which has methods that simply call the **SharedObject<>::Create** method on the appropriate animation class. The lifetime of the animation object depends on the type of animation performed.

The **CarouselPaneMessageHandler** class provides animation for the inner orbital of the carousel and the history stack. The history stack is made up of zero or more history items each of which can be animated. In code, each history item is a **CarouselHistoryItem** object (shown in Listing 9) and this contains a pointer to the animation objects used to animate the item (a **CarouselThumbnailAnimation** object and an **OrbitalAnimation** object). When the user selects a folder it is added to the history stack and a **CarouselHistoryItem** object created for the folder and added to a vector called **m_carouselHistoryItems**. When the user navigates back, the last **CarouselHistoryItem** is removed from the vector, and the animation objects for this folder are destroyed.

Listing 9 History item

C++

```
struct CarouselHistoryItem
{
    ComPtr<IThumbnail> Thumbnail;
    ComPtr<ICarouselThumbnailAnimation> ThumbnailAnimation;
    ComPtr<IOrbitAnimation> OrbitAnimation;
};
```

The inner orbital involves two animations. The first is the rotation of the carousel, which is the rotation of the folder thumbnails about the orbital and is provided by **CarouselAnimation**. The second animation occurs when the user selects a folder and the inner orbital expands to give the impression of zooming into the folder. This animation is provided by an **OrbitalAnimation** object. These two objects are created when the **CarouselPaneMessageHandler** object is first created and their lifetime is the same as the handler object.

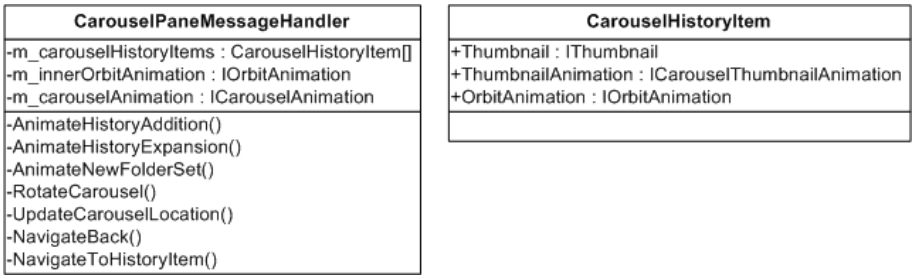
The **MediaPaneMessageHandler** class provides the animation code for the media pane. There are three types of animation that can occur in the media pane: flyer, when the pane is first populated with photos; slide, when you use the arrow buttons to show another page of photos in the pane, and mover, when you resize the window so more or fewer photos are shown in the pane. Only one of these animations can occur at any one time, so the **MediaPaneMessageHandler** object only holds a reference to an object for the current animation. This means that the animation objects (provided by the **FlyerAnimation**, **SlideAnimation** or **MoverAnimation** classes) are only created when needed and live until the next animation.

Animating the Carousel

The **CarouselPaneMessageHandler** class provides the message handler code for the carousel window. This class has objects for the animation of the inner orbital and the history stack. The **m_carouselAnimation** object, an

instance of the **CarouselAnimation** class provides two types of animation for the thumbnails on the inner orbital. The **m_innerOrbitalAnimation** object, an instance of the **OrbitalAnimation** class, provides animation of the ellipse shown for the inner orbital. Instances of the **CarouselThumbnailAnimation** and **OrbitalAnimation** classes are created for each folder on the history stack and provide animation for expanding and contracting the stack. The **CarouselThumbnailAnimation** class provides thumbnail animation and the **OrbitalAnimation** class provides the animation of the orbitals. Figure 5 shows the members of the **CarouselPaneMessageHandler** class.

Figure 5 Showing the animation members of the CarouselPaneMessageHandler class



The first type of animation provided by the **CarouselAnimation** class is when the items in the inner orbital rotate. When you drag one of the folders in the inner orbital the carousel will rotate in the direction you drag the item, but the rotation will continue after you lift your finger. This rotation continues but slows down until the carousel stops spinning. The transition used here is an accelerate-decelerate transition where the entire transition is a deceleration. The pertinent code from **CarouselAnimation::SetupRotation** is shown in Listing 10. The important point is that the third parameter is 0 and the fourth is 1 which means that there is no acceleration. This is a deceleration from the current value to the value represented by the **rotation** parameter, and this occurs over the period of time given by the **duration** parameter. The **rotation** value indicates by how much the carousel will rotate, and since this is an angle, the animation variable attached to the **transition** object will decrease if the rotation is clockwise and increase if the rotation is counter-clockwise. The deceleration refers to the rate of change to get to this final value. When the carousel is drawn the **CarouselPaneMessageHandler::DrawOrbitalItems** method is called and this obtains the rotation value from the **m_carouselAnimation** object and draws the currently selected folder at the specified position on the inner orbital. The **DrawOrbitalItems** method then draws the other items at equal angular positions around the orbital.

Listing 10 Initializing the carousel rotation

C++

```
hr = m_transitionLibrary->CreateAccelerateDecelerateTransition(
    duration,
    rotation,
    0.0,
    1.0,
    &transition);
```

The other animation that the **CarouselAnimation** class provides occurs when the user expands the history stack. In this situation the inner orbital shrinks, moves to the right side and fades; during this animation the folder thumbnails on this orbital shrink to half size and their opacity reduces to 60 percent. When the expanded history list is restacked the inner orbital grows back to full size, moves to the center of the pane, and opacity changes back to 100 percent; during this animation the thumbnails grow back to full size and their opacity changes back to 100 percent. The **CarouselAnimation** class provides two transitions for the size and opacity of the thumbnails, both transitions are linear which means that the size or opacity variable changes at a constant rate. Listing 11 shows the code that does this in the **CarouselAnimation::SetupScale** method and a similar transition is created in the **CarouselAnimation::SetupOpacity** method. Both the **SetupScale** and **SetupOpacity** methods create a storyboard, add the transition to the storyboard, and schedule it. However, the code in **CarouselPaneMessageHandler** always calls these methods as a pairthat means that the two animations will always be performed at the same time: the inner orbital will shrink and get less opaque; or the orbital will grow and get more opaque.

Listing 11 Initializing the carousel scaling

C++

```
hr = m_transitionLibrary->CreateLinearTransition(
    duration,
    thumbnailScale,
    &transition);
```

The carousel pane object has a member **m_innerOrbitalAnimation** which is an instance of the **OrbitalAnimation** class. As the name suggests this object provides the size and position of the inner orbital. When a user selects a folder on the carousel the folder is added to the history stack and if the folder has items, the inner orbital appears to grow from the centre to give the impression of zooming into the folder. This animation is provided by the **m_innerOrbitalAnimation** object which animates the size and opacity of the orbital. The changes are linear, so the **OrbitalAnimation** class creates linear transition objects.

Animating the History Stack

When you click on a folder in the carousel it is added to the history list represented by a vector called **m_carouselHistoryItems** as shown in Figure 5. This vector contains the **CarouselHistoryItem** items shown in Listing 9. The **IThumbnail** reference in the **CarouselHistoryItem** structure is to an object that gives access to the bitmap image and the current position of a thumbnail image. The other two members are used to animate members in the history list and refer to instances of the **CarouselThumbnailAnimation** and **OrbitalAnimation** classes.

When you click on a folder in the carousel, the folder is opened and the subfolders are shown in the inner orbital. At this point two animations occur relevant to the history list. The ellipse representing the orbital containing the folder added to the history list expands to give the impression that the user is zooming into the folder's contents. This animation is performed by the **OrbitalAnimation** class, described earlier. The second animation involves the folder icon moving from the inner orbital to the history stack. This is performed by the **CarouselThumbnailAnimation** class.

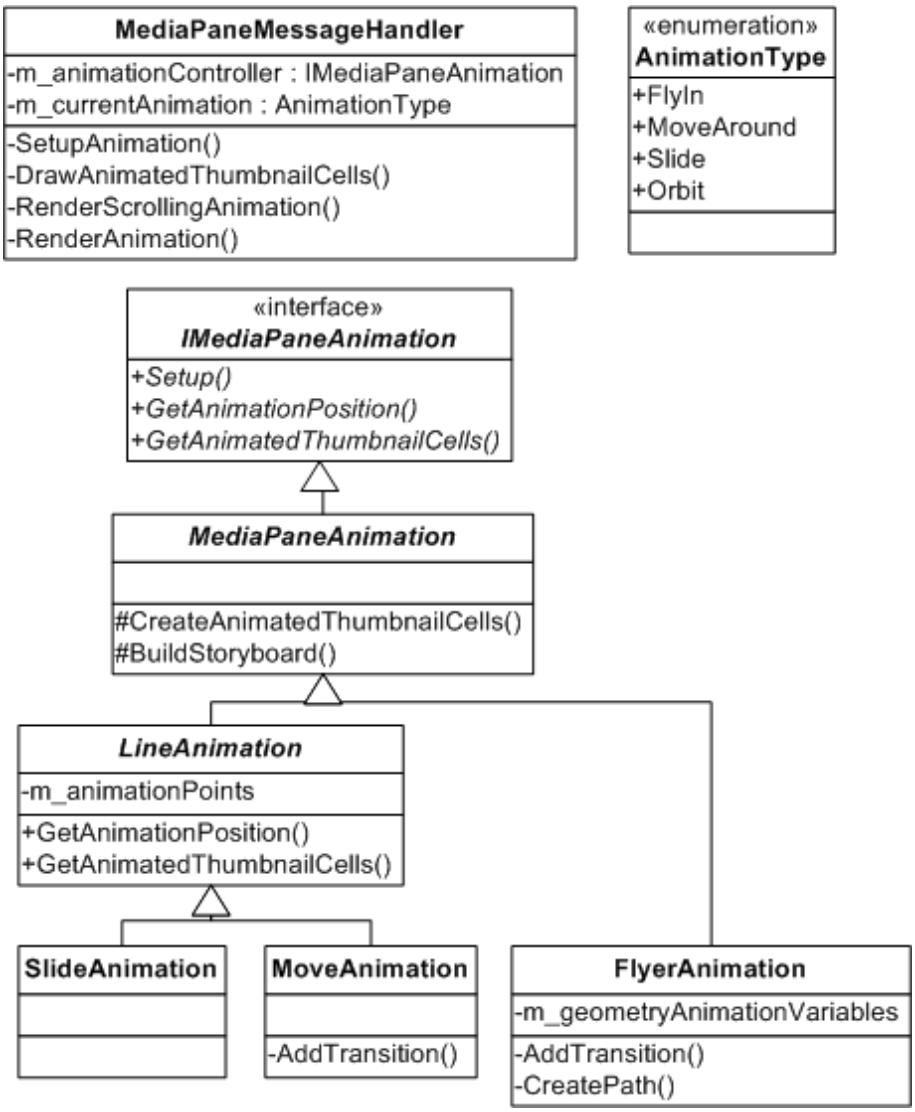
During expansion of the history stack all orbitals are expanded eccentrically, that is, as the orbital expands the center of the ellipse moves. This is performed by the **OrbitalAnimation** class which has five variables, two describe the ellipse dimensions and two describe the ellipse center. The fifth variable describes the opacity of the ellipse. When the stack is expanded or contracted all five of these variables are changed by using linear transitions.

The **CarouselThumbnailAnimation** class provides the animation of thumbnails on the stack. This class encapsulates three variables, the x and y position of the thumbnail, and the opacity. The class provides two types of animation, one has linear transitions for all three variables, but the other provides a accelerate-decelerate transition for all three variables. This latter animation involves first an acceleration and then a deceleration. The way that this is performed is using several transations scheduled using a key frame.

Animating the Media Pane

The media pane uses animation in three situations and there are three concrete classes to provide this: **FlyerAnimation**, **SlideAnimation**, and **MoveAnimation**. The relationship between these classes is shown in Figure 6. Since only one of these animations can be shown at any one time, the media pane message handler class holds a reference to an instance of just one of these classes, **m_animationController**, and the type of the animation is stored in a field called **m_currentAnimation**.

Figure 6 The animation members of the MediaPaneMessageHandler class



The **MediaPaneAnimation** base class has two abstract methods called **CreateAnimatedThumbnailCells** and **BuildStoryboard** that are called in the **Setup** method to create the storyboard specific to the derived class animation. Figure 6 has another abstract class called **LineAnimation** which provides a base implementation to get the position of the animated photo thumbnails.

The derived class implementations of the **BuildStoryboard** method create the storyboard, transitions, and animation variable objects. An instance of each class animates the positions of all the items shown in the media pane, so each such instance has a collection that contains information about each photo, including the animation variables for that particular photo.

The **SlideAnimation** is the simplest of the three animations. This is used when the user views another page of photos and the animation involves the new page of photos sliding into view from either the left or the right. For each photo that is animated there are two animation variables, one each for the x and y coordinates. Since the slide is horizontal, the y coordinate does not change, so the **IUIAnimationTransitionLibrary::CreateConstantTransition** [http://msdn.microsoft.com/en-us/library/dd371903(VS.85).aspx] method is called to create a constant transition. The x coordinate animation variable is associated with an acceleration-deceleration transition where half of the transition is acceleration and the other half a deceleration.

The **MoverAnimation** class provides the animation that rearranges photos in the media pane when the window is resized. The photo thumbnails are rearranged so that the space available is filled with photos and this may mean adding extra rows. The **MoverAnimation** class calculates how the photo will move and creates a variable for the x and y coordinate for each photo. The **AddTransition** private method creates a parabolic transition on one coordinate so that the value decelerates to zero (by calling the **IUIAnimationTransitionLibrary::CreateParabolicTransitionFromAcceleration**

[http://msdn.microsoft.com/en-us/library/dd371934(VS.85).aspx] method), and it creates a accelerate-decelerate transition on the other coordinate where the first 30 percent is an acceleration and the last 30 percent is a deceleration.

The final animation is provided by the **FlyerAnimation** class. This is called when you open a folder, the new photos fly in from the left and any photos in the media pane fly out to the right. Each thumbnail has a Direct2D graphics path defined for it and an animation variable that determines at which point on the path the photo is currently. The graphics path is an object with an **ID2D1Geometry** [http://msdn.microsoft.com/en-us/library/dd316578(v=VS.85).aspx] interface and is an arc. The animation variable determines the position along this arc and changes according to a parabolic transition. This position is converted to actual x-y coordinates y calling the **ID2D1Geometry::ComputerPointAtLength** [http://msdn.microsoft.com/en-us/library/dd316578(v=VS.85).aspx] method.

Conclusion

This article shows how you can create transition objects, which define how a variable changes over time, and how to use a storyboard object to specify the relationship between transitions. It also shows how you can use the animation manager object to obtain the value of the animation objects so that you can draw the animation. Because animation involves redrawing the screen, often at a fast rate, to prevent flicker you should use a graphics API that synchronizes drawing with the monitor refresh, like Direct2D. The Windows Animation Manager and the Direct2D API are the perfect combination to provide animation on Windows 7. The next chapter will cover how Hilo uses the Windows Library API to get access to photo files on the computer.

Chapter 8: Using Windows 7 Libraries and the Shell

Users can store documents, images, and other files in many locations—on different types of hardware installed locally or on other computers on the network. In the past, files were often physically stored in a location according to their type —for example, images in the My Pictures folder, documents in the My Documents folder, and so on. However, a far more powerful and modern way to access files is via their type rather than through their location. This is the purpose of Windows 7 Libraries. Files from many different locations can be accessed through a single logical location according to their type even though they are stored in many different locations. Libraries are user defined collections of content that are indexed to enable faster search and sorting. Hilo uses the Windows 7 Libraries feature to access the user's images.

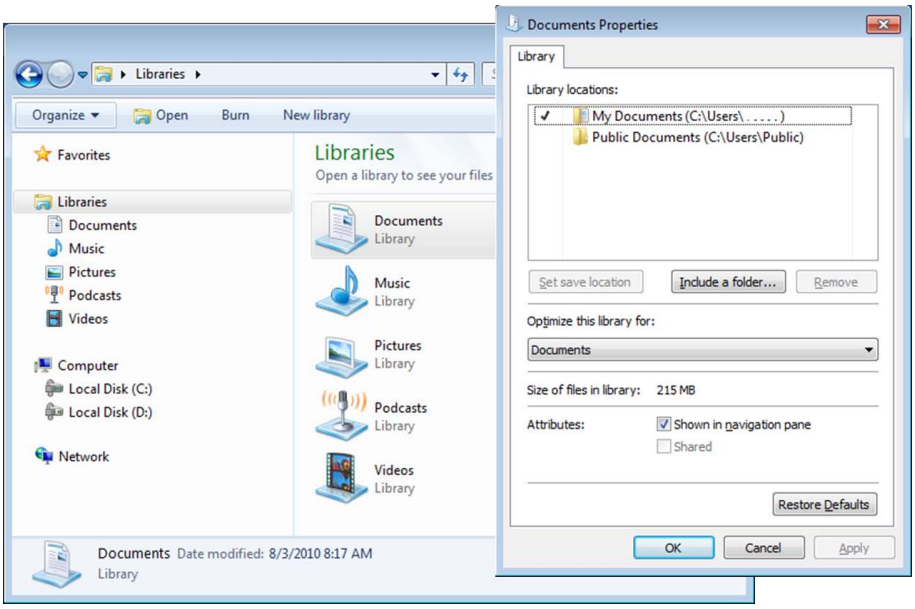
Using the Shell Namespace

The Windows Shell namespace provides access to a wide range of objects through a hierarchical structure. Windows Vista replaced the older constant special item ID list [http://msdn.microsoft.com/en-us/library/bb762494(VS.85).aspx] (CSIDL) naming of special folders with known folder IDs [http://msdn.microsoft.com/en-us/library/dd378457(v=VS.85).aspx] . In both cases the special folders contain specific types of data: video, pictures, or music, but the folders are accessed using an ID that at runtime can give access to the disk storage path. Known folders in Windows Vista offer more features than CSIDL including the ability to change the storage location without changing the applications that rely on the known folder (CSIDL only allows My Documents location to be changed by the user).

Windows 7 extends the idea of known folders with libraries. Windows 7 libraries are user-defined collections of folders, and actions that are performed on the library will be applied to all folders in the library. This means that when you search a Windows 7 library you will search all the folders that are part of the library, and when you stack the items in a library the stacks will contain items from all the folders in the library regardless of the actual location of those folders. Windows 7 indexing is applied to Windows 7 libraries which mean that searches on a library will be performed on all the folders in the library.

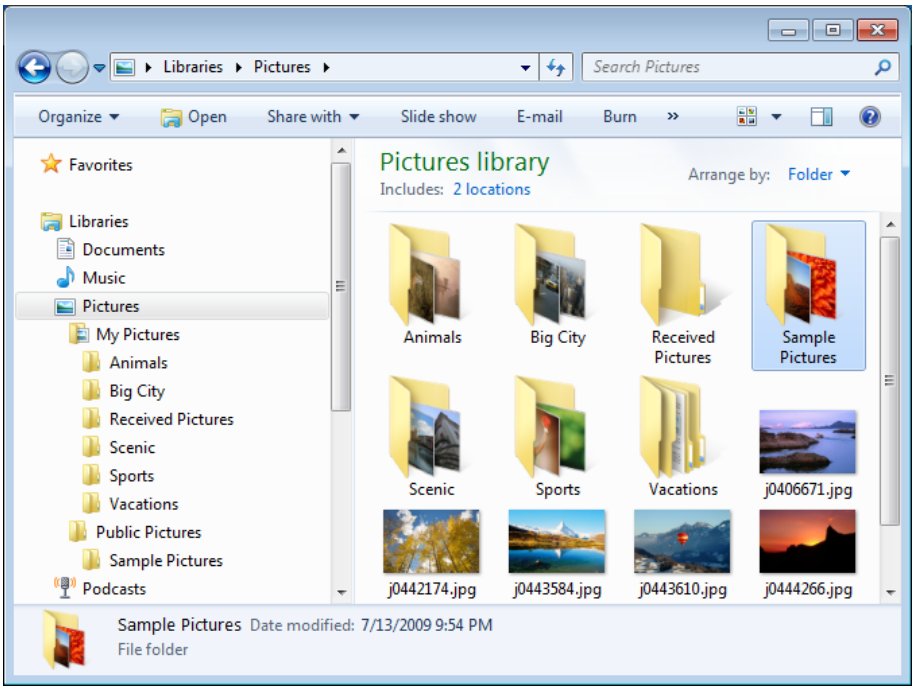
Windows Explorer displays libraries in the navigation pane, as shown in Figure 1. The properties of a library shows the actual folders that will be included. You can use this property dialog to add or remove folders and to determine which folder will be treated as the default save location. You can add any folders on the local disks that your account has access too, and any folders on external drives like USB drives or shares on a server. You cannot add folders on removable drives, nor on remote shares that are not available offline that (MSDN lists the folders that cannot be put in libraries [http://msdn.microsoft.com/en-us/library/dd758095(v=VS.85).aspx#library_managing_folders]).

Figure 1 Windows 7 libraries



When you select a library, Windows Explorer displays an aggregate view of the files and folders that are part of the library, as shown in Figure 2.

Figure 2 Showing the contents of the Documents library



Libraries are logical representations of user content. This means that you store files in the folders that are part of the library and not in the library itself. So for example, the Documents library is the default location for documents and contains the user's documents in their My Documents folder and any documents in the Public Documents folder. Although Windows Explorer displays the Documents library as if it is a folder, no physical folder exists, so if a user saves a file to the Documents library then the file will actually be saved to the default save location (in Figure 1 this save location is set to My Documents).

The Windows 7 Application Programming Interface (API) provides COM-based objects used to access the contents of the libraries. You can traverse through the logical hierarchy through these shell item objects without knowing the absolute storage location paths (although it is possible to obtain the system file path). All items in the shell are represented by an object with the **IShellItem** [http://msdn.microsoft.com/en-us/library/bb761144(v=VS.85).aspx] interface, but specific shell items will implement other interfaces (for example, folder objects also implement the **IShellFolder** [http://msdn.microsoft.com/en-us/library/bb775075(v=VS.85).aspx] interface). It is very important that Windows 7 applications use the Shell API

to access shell items rather than using absolute file system file paths. Equally important is that applications use the Windows 7 common file dialogs because these dialogs show the system's libraries and provide appropriate **IShellItem** objects selected through the dialog.

Using Shell Items

Central to the shell namespace are objects called shell items that implement the **IShellItem** interface. The new common file dialogs (**CLSID_FileOpenDialog** or **CLSID_FileSaveDialog**) refer to items through shell item objects and return an **IShellItem** or an array of such items through the **IShellItemArray** [http://msdn.microsoft.com/en-us/library/bb761106(v=VS.85).aspx] interface. The caller can then use an individual **IShellItem** object to get a file system path or to open a stream object on the item to read or write information, or query for one of the several shell interfaces implemented for specific shell types. The use of **IShellItem** objects is important because these file dialogs can access items in both file system folders and other virtual folders that you find in the shell, including libraries. In addition, Windows 7 provides a new object, **CLSID_ShellLibrary**, specifically to access libraries.

To use the shell API in C++ you include the shobjidl.h header file. This file declares the shell interfaces and the symbols for the CLSIDs for the shell objects that you can create, and it also contains helper methods. Listing 1 shows simple code to create and use a shell item. The **SHCreateItemFromParsingName** [http://msdn.microsoft.com/en-us/library/bb762136(v=VS.85).aspx] method takes a system file path and returns a shell item object. In this example the shell item object is used to obtain a user readable string for the item.

Listing 1 Creating a shell item

C++

```
LPWSTR szFilePath = GetFileNameFromSomewhere(); // Get a file name from somewhere.
IShellItem* pItem = nullptr;
HRESULT hr = ::SHCreateItemFromParsingName(
   szFilePath, nullptr, IID_PPV_ARGS(&pItem));
if (SUCCEEDED(hr))
{
   LPWSTR szName = nullptr;
   hr = pItem->GetDisplayName(SIGDN_NORMALDISPLAY, &szName);
   if (SUCCEEDED(hr))
   {
      wprintf(L"Shell item name: %s\n", szName);
      ::CoTaskMemFree(szName);
   }
   pItem->Release();
}
```

Shell items can represent any object that can be displayed in the shell: a file, a folder, a shortcut, or even virtual folder items like the Recycle Bin, and libraries. You can get information about the type of item by calling the **IShellItem::GetAttributes** [http://msdn.microsoft.com/en-us/library/bb761138(v=VS.85).aspx] method. Listing 2 shows code that accesses the attributes of a shell object. In this case the **SHCreateItemInKnownFolder** [http://msdn.microsoft.com/en-us/library/bb762136(v=VS.85).aspx] method is called to get a shell object for the libraries on the computer. This method creates the shell item object, it does not create the files or folders that the shell item object refers to. The first parameter of this method is a GUID for the known folder that is one of the values defined in knownfolders.h. The third parameter is the name of the item within the known folder that you want to access, or if the parameter is null (as in this case) the shell item object will be to the known folder. Listing 2 accesses the Libraries folder and as mentioned above, this folder does not physically exist on a computer, instead the shell item gives access to other libraries in the shell namespace.

Listing 2 Accessing a known folder

C++

```
IShellItem* pItem = nullptr;
HRESULT hr = ::SHCreateItemInKnownFolder(
   FOLDERID_Libraries, 0, nullptr, IID_PPV_ARGS(&pItem));
if (SUCCEEDED(hr))
```

```
{
   DWORD dwAttr = 0;
   hr = pItem->GetAttributes(SFGAO_FILESYSANCESTOR, &dwAttr);
   if (SUCCEEDED(hr))
   {
      if (SFGAO_FILESYSANCESTOR == dwAttr)
      {
         wprintf(L"Item is a file system folder\n");
      }
   }
   pItem->Release();
}
```

You can also use the **SHGetKnownFolderItem** [http://msdn.microsoft.com/en-us/library/dd378429(VS.85).aspx] function to get a shell item for a known folder. This function also allows you to pass an access token so that you can access known folders restricted to users other than the current logged on user.

Accessing Shell Item Properties

You can get additional information about a shell item by requesting the value of an item property. To do this you should use the methods on the **IShellItem2** [http://msdn.microsoft.com/en-us/library/bb761130(v=VS.85).aspx] rather than **IShellItem**. Each property is identified by the values in a **PROPERTKEY** [http://msdn.microsoft.com/en-us/library/bb773381(VS.85).aspx] structure and propkey.h defines the initialized values for the properties that you can request. Properties can be strings, numeric values, or dates, and there are methods on **IShellItem2** to return appropriate values. For example, Listing 3 shows how to access the date that the item was created by accessing the **PKEY_DateCreated** property and it assumes the **pItem** variable has already been assigned to an **IShellItem** object.

Listing 3 Accessing a property through a shell item

C++

```
IShellItem2* pItem2 = nullptr;
hr = pItem->QueryInterface(&pItem2);
if (SUCCEEDED(hr))
{
   FILETIME ft = {0};
   pItem2->GetFileTime(PKEY_DateCreated, &ft);
   SYSTEMTIME st = {0};
   ::FileTimeToSystemTime(&ft, &st);
   wprintf(
      L"Date Created: %04d-%02d-%02d %02d:%02d:%02d\n",
      st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond);
   pItem2->Release();
}
```

Binding to Handler Objects

The methods of the **IShellItem** and **IShellItem2** interfaces give limited access to the shell object, however, you can obtain a handler object to get additional access by calling the **IShellItem::BindToHandler** [http://msdn.microsoft.com/en-us/library/bb761134(v=VS.85).aspx] method. There are many types of handler objects and different shell items will use different handler objects. If the item is a file then you can access a **IStream** handler object, if the item is a folder then you can access an **IShellFolder** handler. The **BindToHandler** method is passed a GUID (defined in shlguid.h) to the type of the handler object you want to obtain. and the method returns a pointer to the handler object. So assuming that the **pItem** variable is the shell item for the Libraries folder initialized in Listing 2, the code in Listing 4 obtains an enumerator object to enumerate the child items and print their names.

Listing 4 Enumerating items in a folder shell item

C++

```
IEnumShellItems* pEnum = nullptr;
```

```
hr = pItem->BindToHandler(nullptr, BHID_EnumItems, IID_PPV_ARGS(&pEnum));
if (SUCCEEDED(hr))
{
   IShellItem* pChildItem = nullptr;
   ULONG ulFetched = 0;
   do
   {
      hr = pEnum->Next(1, & pChildItem, &ulFetched);
      if (FAILED(hr)) break;
      if (ulFetched != 0)
      {
         LPWSTR szChildName = nullptr;
         child->GetDisplayName(SIGDN_NORMALDISPLAY, &szChildName);
         wprintf(L"Obtained %s\n", szChildName);
         CoTaskMemFree(szChildName);
         pChildItem ->Release();
      }
   } while (hr != S_FALSE);
   pEnum->Release();
}
```

If the shell item is a file, it is up to you as the developer to determine how to load the data from the file. If you wish to use the Windows API functions like **ReadFile** [http://msdn.microsoft.com/en-us/library/aa365467(VS.85).aspx] to read from the file then you must obtain a handle to the file. To do this you can pass the name returned from **IShellItem::GetDisplayName** for the **SIGDN_FILESYSPATH** name type to the **CreateFile** [http://msdn.microsoft.com/en-us/library/aa363858(v=VS.85).aspx] function. You can also open the shell item as a stream object by requesting the **BHID_Stream** handler in a call to the **IShellItem::BindToHandler** method.

Using Common File Dialogs

In addition, other APIs will act upon shell item objects, for example you can use the Common File Dialog to obtain a shell item with the path to where you want a file saved. The Common File Dialog object does not open or save a file, instead it gives information about the shell item that the user identifies. Listing 5 shows the basic code to save a file, here the **pInitialItem** variable is an initialized shell item object that indicates the file that is initially shown in the Save As dialog by calling the **IFileSaveAsDialog::SetSaveAsItem** [print-Hilo-2011_file:///C:/Users/RoAnnC/AppData/Local/Microsoft/Windows/Temporary Internet Files/Content.Outlook/36ZE61SY/microsoft.com/en-us/library/bb775712(v=VS.85).aspx] method. The **IFileSaveDialog::Show** [http://msdn.microsoft.com/en-us/library/bb761688(v=VS.85).aspx] method displays the dialog and if the user clicks the **OK** button this method will return **S_OK**, otherwise if the user clicks the **Cancel** button then the dialog will return **ERROR_CANCELLED**. If the user has specified a file in the dialog then a shell item with information about this file is obtained through a call to **IFileSaveDialog::GetResult** [http://msdn.microsoft.com/en-us/library/bb775964(v=VS.85).aspx] . In this case, the shell item object will not necessarily reference an actual file, the shell item object simply contains the path and file name provided by the user in the dialog. The code then has to provide its own code to copy the file referenced by the **pInitialItem** shell item to the location specified by the **pSaveAsItem** shell item.

Listing 5 Using the Save As dialog

C++

```
IFileSaveDialog* pShellDialog;
hr = CoCreateInstance(
   CLSID_FileSaveDialog, nullptr, CLSCTX_INPROC, IID_PPV_ARGS(&pShellDialog));
if (SUCCEEDED(hr))
{
   pShellDialog->SetSaveAsItem(pInitialItem);
   pShellDialog->Show(nullptr);
   if (SUCCEEDED(hr))
   {
      IShellItem* pSaveAsItem = nullptr;
      hr = pShellDialog->GetResult(&pSaveAsItem);
```

```
        if (SUCCEEDED(hr))
        {
            CopyFileTo(pInitialItem, pSaveAsItem); // Call the code to Copy copy the actual file…
            pSaveAsItem->Release();
        }
    }
    else
    {
        // If the user clicks cancel the return value is 0x800704c7, that is
        // HRESULT_CODE(hr) == ERROR_CANCELLED
    }
    pShellDialog->Release();
}
```

The Save As dialog shown in Listing 5 will also show the Libraries folder in the navigation pane. If the user selects a library then the shell item returned from the **IFileSaveDialog::GetResult** method will reference the actual file system location, if necessary using the default save location for the library.

Using the Shell Library Object

The Windows 7 API provides a COM object to allow you to administer libraries. The shell library object implements the **IShellLibrary** [http://msdn.microsoft.com/en-us/library/dd391719(v=VS.85).aspx] interface and the Windows API provides a helper method **SHCreateLibrary** [http://msdn.microsoft.com/en-us/library/dd378433(VS.85).aspx] that creates an uninitialized object. You can call **IShellLibrary::LoadLibraryFromKnownFolder** [http://msdn.microsoft.com/en-us/library/dd391721(v=VS.85).aspx] to initialize the object to refer to a known folder. The shell library object can be used to add or remove folders from the library, enumerate the folders in the library, and set the default save folder. The library object is used to write to the description file (library-ms file) for the library, and once you have finished changing the library settings you must call the **IShellLibrary:Commit** [http://msdn.microsoft.com/en-us/library/dd391711(v=VS.85).aspx] method if the library already exists, or the **IShellLibrary::Save** [http://msdn.microsoft.com/en-us/library/dd391724(v=VS.85).aspx] if this is a new library.

Using Windows 7 Libraries in Hilo

The carousel and media panes in the Hilo Browser show thumbnail representations of folders and photos, so the message handlers for these classes need to be initialized with information about these items. In Hilo this is done through a struct called **ThumbnailInfo** that has a member which is a reference to an **IShellItem** object. When a user selects a folder in the carousel, the window message handler enumerates the items in the folder and displays the subfolders in the inner orbital and the photos in the media pane. To do this, Hilo has to enumerate and filter shell items.

Initializing the Browser Carousel

The **BrowserApplication** class has a variable called **m_currentBrowseLocationItem** that is a shell item to the initial folder shown by the Browser carousel. The variable is initialized when the Browser is started with the code shown in Listing 6. The first part of this code calls the **SHCreateItemInKnownFolder** method to get a shell item object for the Pictures library and if this call is not successful the code then calls **SHGetKnownFolderItem** to obtain the shell item for the Computer folder which gives access to all the hard drives (including external drives) accessible by the computer.

Listing 6 Determining the initial folder for the carousel

C++

```
// No location has been defined yet, so we just load the pictures library
if (nullptr == m_currentBrowseLocationItem)
{
    // Default to Libraries library
    hr = ::SHCreateItemInKnownFolder(
        FOLDERID_PicturesLibrary, 0, nullptr, IID_PPV_ARGS(&m_currentBrowseLocationItem));
}
```

```
// If the Pictures Library was not not found
if (FAILED(hr))
{
   // Try obtaining the "Computer" known folder
   hr = ::SHGetKnownFolderItem(
      FOLDERID_ComputerFolder, static_cast<KNOWN_FOLDER_FLAG>(0), nullptr,
      IID_PPV_ARGS(&m_currentBrowseLocationItem));
}

if (SUCCEEDED(hr))
{
   ComPtr<IPane> carouselPane;
   hr = carouselPaneHandler.QueryInterface(&carouselPane);

   if (SUCCEEDED(hr))
   {
      hr = carouselPane->SetCurrentLocation(m_currentBrowseLocationItem, false);
   }
}
```

The message handler class for the carousel pane is updated with the current location by passing the shell item to the **SetCurrentLocation** method. This method enumerates the folders in the selected folder and updates the carousel to show these subfolders. The carousel pane handler class then calls the **SetCurrentLocation** method on the media pane which enumerates the image files in the selected folder and uses this to populate the thumbnails in the media pane.

Whenever the user selects a folder in the carousel or in the history list the shell item for the newly selected item is passed to **SetCurrentLocation** of the carousel handler and hence the carousel and media pane are updated with the items in the selected folder.

Enumerating Folders

Hilo provides a utility class (in the Common project) called **ShellItemsLoader**. This class has one public static method called **EnumerateFolderItems** that is passed the shell item object of the folder to enumerate, a value indicating if the method should return the folders or image file items in the folder, and a parameter indicating whether the search is recursive or not. This method returns a vector of the shell item objects for the requested items.

Listing 7 shows the first part of this method, **1** indicates the type of objects to search for and is used to add named values to the **itemKinds** vector. When the method enumerates items in the folder it obtains the type of each item, and if the type of the item is one of those in the **itemKinds** vector the item is added to the **shellItems** vector returned to the caller.

Listing 7 Enumerating folders: initialization code

C++

```
HRESULT ShellItemsLoader::EnumerateFolderItemsNonRecursive(
   IShellItem* currentBrowseLocation, ShellFileType fileType,
   std::vector<ComPtr<IShellItem> >& shellItems)
{
    std::vector<std::wstring> itemKinds;

    if ((fileType & FileTypeImage) == FileTypeImage)
    {
        itemKinds.push_back(L"picture");
    }

    if ((fileType & FileTypeImage) == FileTypeVideo)
    {
        itemKinds.push_back(L"video");
    }
```

```
    if ((fileType & FileTypeImage) == FileTypeAudio)
    {
        itemKinds.push_back(L"music");
    }

    // Followed by code to do the enumeration
```

The folder is enumerated by using a **IShellFolder** [http://msdn.microsoft.com/en-us/library/bb775075(v=VS.85).aspx] handler object and Listing 8 shows the code that obtains this object by calling the **BindToHandler** method.

Listing 8 Enumerating folders: creating the handler object

C++

```
    // Enumerate all objects in the current search folder
    ComPtr<IShellFolder> searchFolder;
    HRESULT hr = currentBrowseLocation->BindToHandler(
        nullptr, BHID_SFObject, IID_PPV_ARGS(&searchFolder));
    if (SUCCEEDED(hr))
    {
        // Enumeration code, see Listing 9
    }
```

The **IShellFolder** object has an enumeration object that implements the **IEnumIDList** [http://msdn.microsoft.com/en-us/library/bb761982(v=VS.85).aspx] interface, this interface enumerates item IDs rather than shell items, but it is possible to create a shell item from an ID by calling the **SHCreateItemWithParent** [http://msdn.microsoft.com/en-us/library/bb762137(v=VS.85).aspx] method. Listing 9 shows the main code that enumerates items in the specified folder. First the code initializes a **SHCONTF** [http://msdn.microsoft.com/en-us/library/bb762539(VS.85).aspx] flag to indicate whether the searched items should be folders or files. This flag is passed to the **IShellFolder::EnumObjects** [http://msdn.microsoft.com/en-us/library/bb775066(v=VS.85).aspx] method that returns an enumeration object with the items. The code then repeatedly calls **IEnumIDList::Next** [http://msdn.microsoft.com/en-us/library/bb761983(v=VS.85).aspx] on this object to obtain the item's ID and calls the **SHCreateItemWithParent** method to create the shell item object.

Listing 9 Enumerating folders: enumerating items

C++

```
        bool const isEnumFolders = (fileType & FileTypeFolder) == FileTypeFolder;
        SHCONTF const flags = isEnumFolders ? SHCONTF_FOLDERS : SHCONTF_NONFOLDERS;

        ComPtr<IEnumIDList> fileList;
        if (S_OK == searchFolder->EnumObjects(nullptr, flags, &fileList))
        {
            ITEMID_CHILD* idList = nullptr;
            unsigned long fetched;
            while (S_OK == fileList->Next(1, &idList, &fetched))
            {
                ComPtr<IShellItem2> shellItem;
                hr = SHCreateItemWithParent(nullptr, searchFolder, idList,
IID_PPV_ARGS(&shellItem));
                if (SUCCEEDED(hr))
                {
                    // Check to see if the item should be added to the returned item vector
                    // See Listing 10
                }

                ILFree(idList);
            }
        }
```

```
    }
    return hr;
```

The shell item created in Listing 9 will either be for a folder or a nonfolder item. If the search is for folders then no further processing is necessary and the item is added to the **shellItems** vector. If the item is a nonfolder object, then the code must check to see if the item is one of the types of files requested, this code is shown in Listing 10. This code reads the **PKEY_Kind** property of the item to obtain the item type as a string and compares the returned value with the items in the **shellItems** vector.

Listing 10 Enumerating folders: checking the item type

C++

```
                    if (isEnumFolders)
                    {
                        shellItems.push_back(static_cast<IShellItem*>(shellItem));
                    }
                    else
                    {
                        // Check if we the item is correct
                        wchar_t *itemType = nullptr;
                        hr = shellItem->GetString(PKEY_Kind, &itemType);
                        if (SUCCEEDED(hr))
                        {
                            auto found = std::find(itemKinds.begin(), itemKinds.end(),
itemType);
                            if (found != itemKinds.end())
                            {
                                shellItems.push_back(static_cast<IShellItem*>(shellItem));
                            }
                            ::CoTaskMemFree(itemType);
                        }
                    }
```

Conclusion

In this chapter you have seen how to use the shell API to access folders and items through Windows 7 libraries. In the next chapter, we will introduce the Annotator application which allows the user to easily edit their images.

Chapter 9: Introducing Hilo Annotator

Hilo is a collection of sample applications that allow you to browse, annotate, and share photographs and images. The previous articles in this series described the design and implementation of the Hilo Browser application, which allows you to browse and select images using a touch-enabled user interface. This article describes the Hilo Annotator application, which allows you crop, rotate, and draw on the photographs you have selected. Hilo Annotator uses the Windows Ribbon Control to provide easy access to the various annotation functions, and the Windows Imaging Component to load and manipulate the images and their metadata.

Integrating the Browser and the Annotator

The Hilo Annotator is a separate application that you can launch either directly from the desktop, the command line, or from within the Hilo Browser application itself. The Browser application was updated to support the integration of the Annotator. The user can launch the Annotator from within the Browser by double tapping on a photo with their finger or double-clicking with the mouse. This action generates a **WM_LBUTTONDBLCLK** [http://msdn.microsoft.com/en-us/library/ms645606(v=VS.85).aspx] message which is handled by the media pane through the **MediaPaneMessageHandler::LaunchAnnotator** method, by passing the name of the selected photo. Listing 1 shows the code for the **LaunchAnnotator** method.

Listing 1 Hilo Browser code to launch the Annotator process

C++

```
void MediaPaneMessageHandler::LaunchAnnotator(std::wstring fileName)
{
    // Get the path of target exe first
    wchar_t currentFileName[FILENAME_MAX];

    if ( !GetModuleFileName(nullptr, currentFileName, FILENAME_MAX) )
    {
        return;
    }

    // Annotator should be found in the same directory as this binary
    std::wstring currentDirectory = std::wstring(currentFileName);
    std::wstring externalFileName = currentDirectory.substr(
                                0, currentDirectory.find_last_of(L"\\") + 1);
    externalFileName += L"annotator.exe";

    STARTUPINFO startInfo;
    PROCESS_INFORMATION processInfo;

    // Initialize startup and process info structures
    ZeroMemory(&startInfo, sizeof(startInfo));
    startInfo.cb = sizeof(startInfo);
    ZeroMemory(&processInfo, sizeof(processInfo));

    // Create command line parameter list
    wchar_t buffer[FILENAME_MAX];
    swprintf_s(buffer, FILENAME_MAX, L"\"%s\" \"%s\"", externalFileName.c_str(),
               fileName.c_str());

    ::CreateProcess(nullptr, buffer, nullptr, nullptr, false, 0, nullptr, nullptr,
                    &startInfo, &processInfo);

    // Release memory
```

```
    ::CloseHandle(processInfo.hProcess);
    ::CloseHandle(processInfo.hThread);

    return;
}
```

The Browser assumes that the Annotator executable is in the same folder as the Browser. The first part of the **LaunchAnnotator** method gets the full path to the Browser application by calling the **GetModuleFileName** [http://msdn.microsoft.com/en-us/library/ms683197(VS.85).aspx] function and then extracts the folder path. This folder path is used as the path to the Annotator application.

The Browser then launches the Annotator by using the **CreateProcess** [http://msdn.microsoft.com/en-us/library/ms682425(VS.85).aspx] function, and specifies the name of the photo to edit via the command line. The Annotator process accesses the command line in the **AnnotatorApplication::Initialize** method when the application first starts. Listing 2 shows the code to do this. First the code calls the **GetCommandLineW** [http://msdn.microsoft.com/en-us/library/ms683156(VS.85).aspx] method and then splits this into an array of pointers to the individual command line arguments by calling the **CommandLineToArgvW** [http://msdn.microsoft.com/en-us/library/bb776391(v=VS.85).aspx] function. The command line is part of the process environment, so the process does not need to provide storage for the string, nor provide code to de-allocate the string buffer. The **GetCommandLineW** function is used because the **lpCmdLine** parameter of the process entry point function, **WinMain** [http://msdn.microsoft.com/en-us/library/ms633559(VS.85).aspx] , can *only* provide the command line as an ANSI string even if, as in the case of Hilo, the process is compiled for Unicode.

Listing 2 Annotator code to access the command line parameters

C++

```
int argumentCount = 0;
ComPtr<IShellItem> currentBrowseLocationItem;
if (SUCCEEDED(hr))
{
   // Process command line
   wchar_t ** commandArgumentList = CommandLineToArgvW(
                                          GetCommandLineW(), &argumentCount);

   if (argumentCount > 1)
   {
      hr = ::SHCreateItemFromParsingName(
                     commandArgumentList[1],
            nullptr,
            IID_PPV_ARGS(&currentBrowseLocationItem));
   }
   else
   {
      // Default to pictures library
      hr = ::SHCreateItemInKnownFolder(
         FOLDERID_PicturesLibrary, 0, nullptr,
         IID_PPV_ARGS(&currentBrowseLocationItem));

      if (FAILED(hr))
      {
         // Set to top-level computer folder
         hr = ::SHGetKnownFolderItem(
                 FOLDERID_ComputerFolder, static_cast<KNOWN_FOLDER_FLAG>(0),
                 nullptr, IID_PPV_ARGS(&currentBrowseLocationItem));
      }
   }
}
```

Listing 2 also shows that if the process is started with a command line parameter then it is used to create a shell item object by calling the **SHCreateItemFromParsingName** [http://msdn.microsoft.com/en-

us/library/bb762134(VS.85).aspx] function. If the Annotator is called without a command line parameter, the Annotator obtains as the starting point the Pictures library, or failing that, the Computer folder. It is important to note that the shell item object can either be a file (the photo passed by the Browser) or a folder (Pictures library or Computer folder). Annotator uses this shell item to populate the editor pane (the equivalent of the Browser's media pane) with all the photos in the specified folder, or if the shell item object is a file, the selected photo.

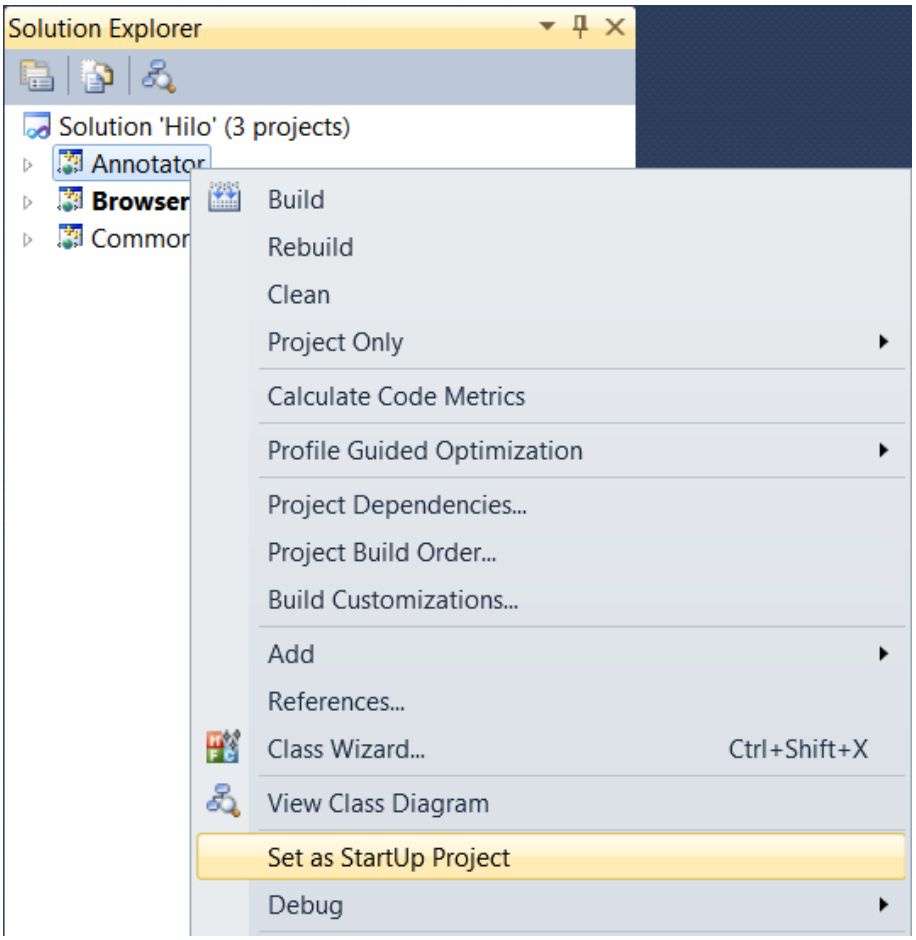
Other than the command line there is no other communication between Browser and Annotator. The Annotator is a separate process so the user can task switch to the Browser and continue to use it. The user can also create another instance of the Annotator. Note that since there is no direct communication with the Browser application if you change a photo in Annotator the cached image of the photo in Browser is not automatically updated to show the changes made in Annotator.

Debugging the Annotator

The multi-process architecture of the Browser and Annotator applications has implications for how you debug the overall solution. Since the Annotator is a separate process, if you are debugging the Browser and launch the Annotator, you cannot step into the Annotator code. Similarly, if you are debugging the Browser you cannot set breakpoints in the Annotator. Instead, if you wish to debug the Annotator you have three options.

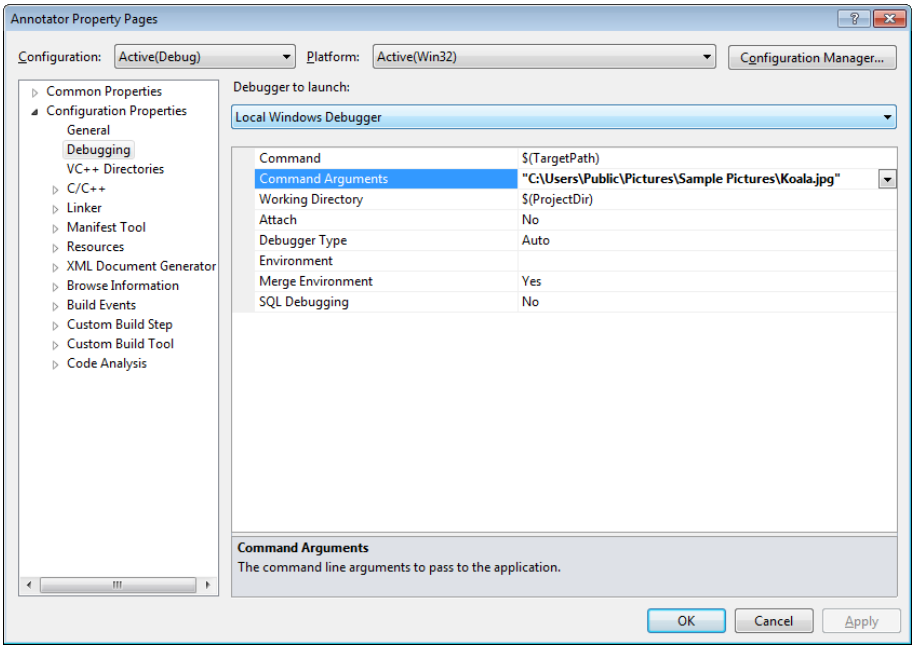
First, you may specify that the Annotator process is started for debugging by clicking the **Set as StartUp Project** menu item in Solution Explorer (Figure 1). This option is useful since it allows you to place breakpoints anywhere in the process, including the **WinMain** function.

Figure 1 Setting the Annotator as the StartUp Project



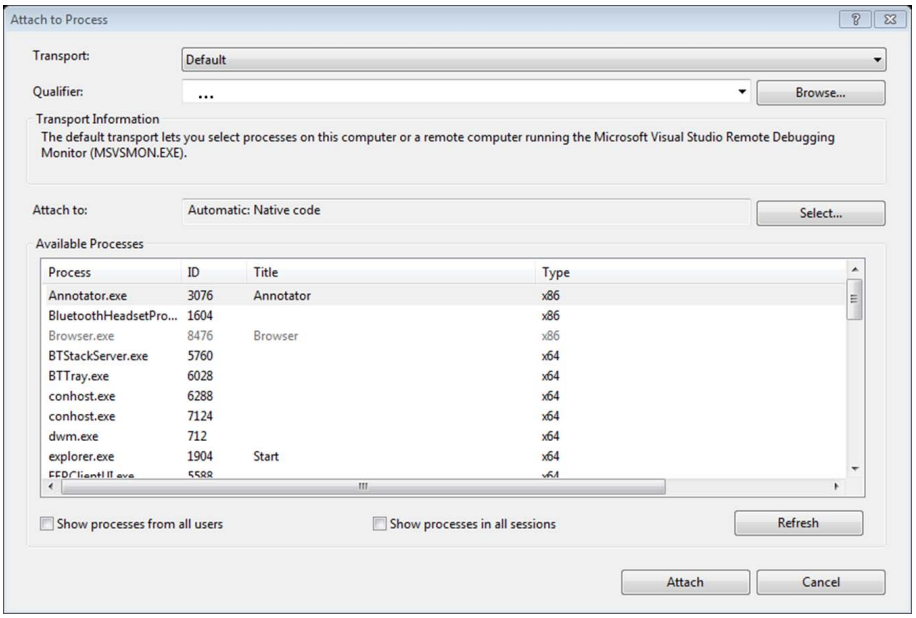
The Annotator process will be started by, and run under, the debugger. This means that because it is not started by the Browser it will not have the file path passed to it by the Browser. If you want to test how the Annotator handles command line parameters then you have to give the full path to a photo as the **Command Arguments** property on the Annotator **Debugging** property page, as shown in Figure 2.

Figure 2 Specifying an image on the Annotator command line



Second, you may allow the Browser to start the Annotator process and then attach to the Annotator process with the Visual Studio debugger. To do this, select the **Attach to Process** menu item on the **Debug** menu. The **Attach to Process** dialog lists all the running processes on the computer. To attach the debugger double-click on the line for **Annotator.exe** as shown in Figure 3. This option is useful for attaching to any existing process, but any debugging can only be done from the point that you attach, which usually means that you cannot debug the **WinMain** function nor the code that creates and initializes the window.

Figure 3 Attaching the debugger to the Annotator process

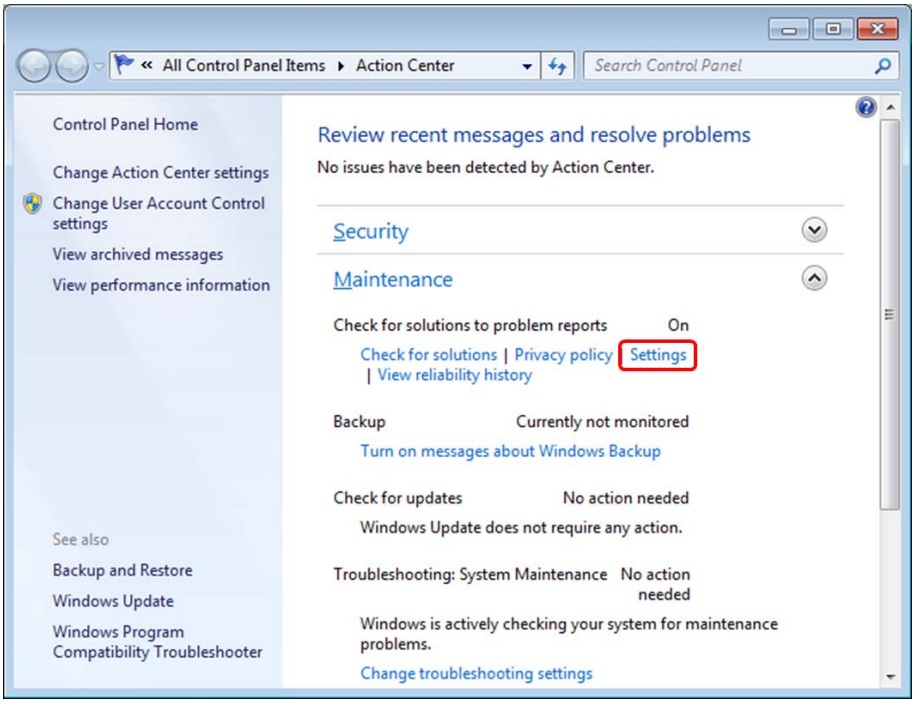


The third option is not available for Visual C++ Express but is available for Visual Studio Professional and above: use Just In Time (JIT) debugging. To do this you put a call to the **DebugBreak** [http://msdn.microsoft.com/en-us/library/ms679297(VS.85).aspx] function (or the **__debugbreak** [http://msdn.microsoft.com/en-us/library/f408b4et.aspx] intrinsic) at the point in your program where you want debugging to start. However, before you can use these functions you have to tell Windows 7 to allow the function call to start the debugger. These functions cause a software exception (an interrupt, **int 3**) and by default, Windows 7 security will treat all exceptions as faults in the program and will handle this by searching for a solution online, so you must disable this action for the Annotator process. If the **assert** C runtime library (CRT) function is called with a false condition the **__debugbreak** [http://msdn.microsoft.com/en-us/library/f408b4et.aspx] intrinsic is called, so if you have asserts in your code and you want them handled through JIT debugging you must disable Windows 7 problem

solving as explained below.

To disable problem solving open the **Action Center** in the **Control Panel**, expand the **Maintenance** section (Figure 4), and click on the **Settings** link.

Figure 4 Using the Action Center



This shows the **Problem Reporting Settings** page which lists the settings that will be used for all processes running on the computer (Figure 3). This dialog box also allows you to list the programs that will be excluded from problem reporting. Click on the **Select programs to exclude from reporting** link and then use the **Add** button to locate and select the debug build of the Annotator process (Figures 5 and 6).

Figure 5 Using the Problem Reporting Settings dialog

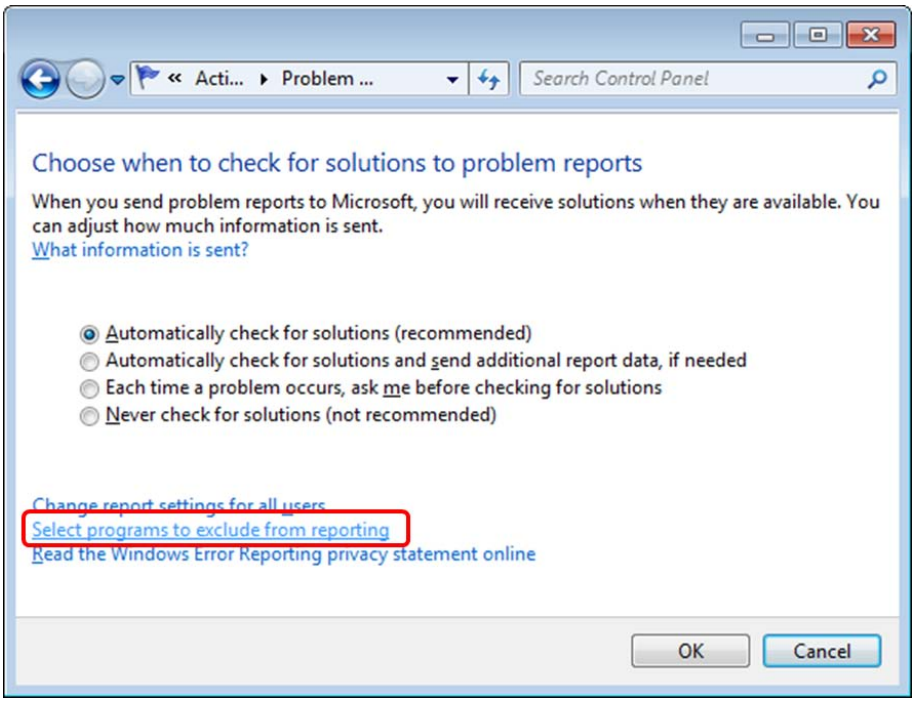
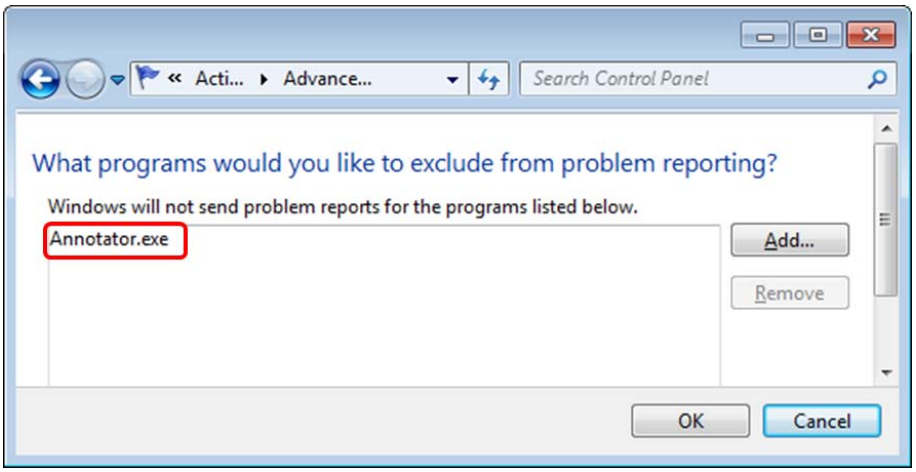
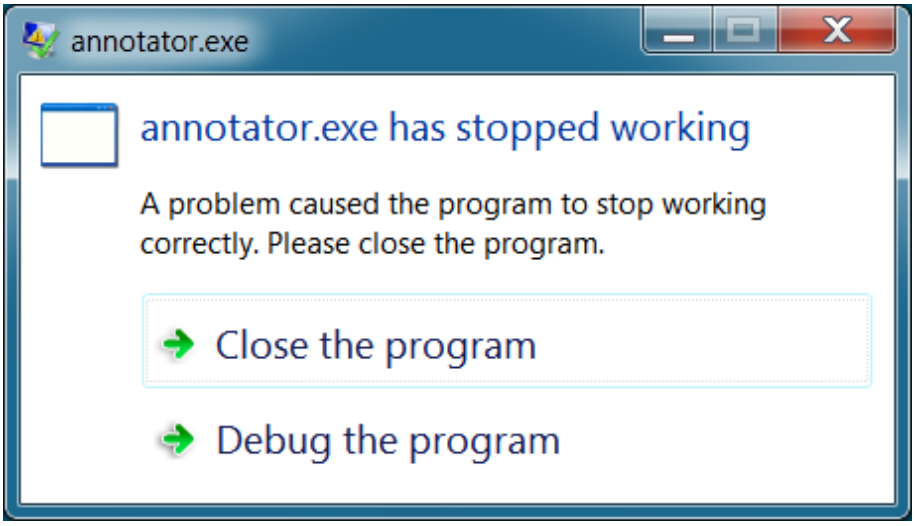


Figure 6 Excluding the Annotator process from problem reporting



Now whenever Annotator is run and there is a call to the **DebugBreak** function, Windows 7 will give you one or more dialog boxes similar to Figure 7. Then it will call the Visual Studio JIT Debugger, which will give you the option to start a new instance of Visual Studio or attach the debugger from a running instance.

Figure 7 Windows 7 problem reporting allowing you to debug a process

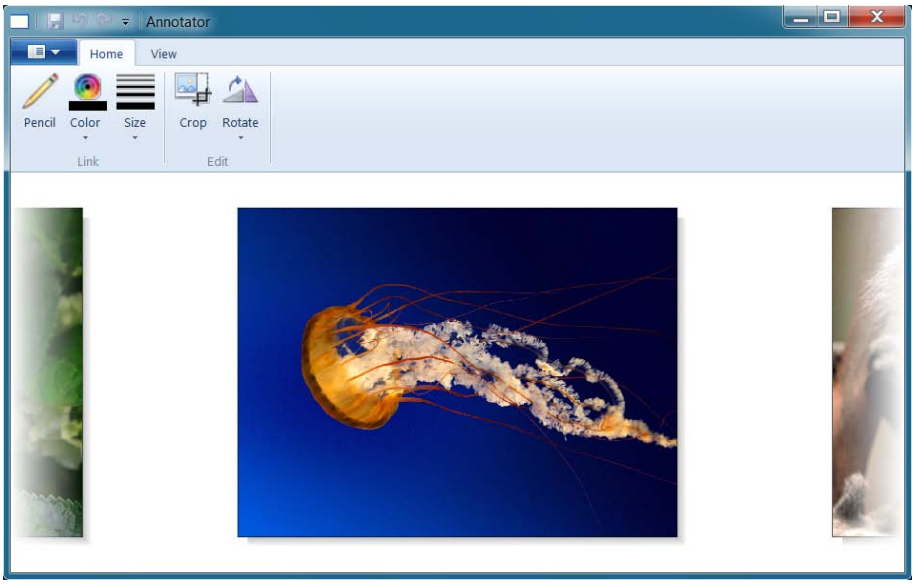


The advantage of JIT debugging is that you can debug any code where you can put a call to the **DebugBreak** function, however, you must make sure that you remove this code when you have finished testing the application.

Examining the Annotator UI

The Annotator process is started by double tapping a photo in the media pane of the Browser with your finger, or by double-clicking with the mouse. The Hilo Annotator user interface is shown in Figure 8. There are three main areas to consider: the image editor, the Ribbon, and the title bar.

Figure 8 The Hilo Annotator user interface



The image editor area takes up most of the application's window. It behaves in a similar way to the media pane in the Browser. You can scroll left or right by using the mouse, the left and right arrow keys, or by dragging a photo with your finger on a touch screen. The default zoom level shows one complete photo and a preview of the photos to the left and right. The Annotator fades the left and right photos by first rendering the images and then drawing over them with a white linear gradient brush (where the gradient is the alpha channel changing from fully opaque to transparent).

Above the image editor is a Windows Ribbon control. This ribbon has two tabs, a menu, and a quick access toolbar. The **Home** tab has two groups and these have controls that allow you to crop, rotate, and draw on the photo in the image list. When you click either the **Pencil** or **Crop** buttons it selects the appropriate action that either allows you to draw with a pencil or crop the image. When you click the **Rotate** button the Annotator rotates the photo clockwise. The **Rotate** control also includes a dropdown menu giving additional transformation options: rotate counter-clockwise, mirror horizontally, or mirror vertically. The **Color** control is a standard control called a **Dropdown Color Picker**. When you click on this control a color swatch is displayed. The **Size** control is a drop down list that displays the four different pencil widths that are available. The **View** tab has three push button controls: zoom in, zoom out, and reset to 100% zoom.

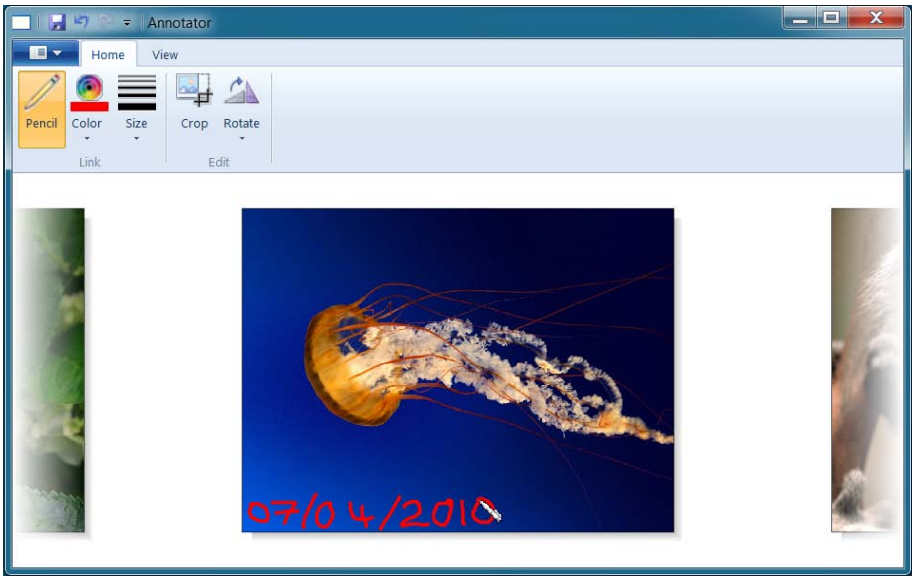
The ribbon has two menus. The main menu is a dropdown menu control that is to the left of the **Home** ribbon tab. This menu has the following items: **Open**, **Save**, **Save A Copy As**, and **Exit.** The other menu is the Quick Access Toolbar and by default this is shown on the title bar. The Quick Access Toolbar has four controls. The first three are buttons that generate commands to save the photo, to undo an action, or redo an action. The fourth control is a dropdown menu that allows you to customize the toolbar: show or hide the other buttons, change the location of the toolbar, and minimize the ribbon.

The default position of the Quick Access Toolbar is on the title bar, but the customize menu has a menu item called **Show below the Ribbon**, when you click this item the toolbar moves to beneath the ribbon control, and the image list is resized accordingly. Clearly moving the toolbar to beneath the ribbon means that the area occupied by the photos is reduced. To give the image editor additional space you can select the **Minimize the Ribbon** item on the quick access toolbar. When the ribbon is minimized only the tab headers are shown. If you click on one of these headers the tab appears, but in front of the photo rather than above it.

Using the Annotator

When you start Annotator from the **Start** menu as a standalone process, you can use the **Open** menu item to select the photo to edit. When you double tap on a photo in the Browser it will start Annotator and the photo selected in the Browser will be opened for editing. You can draw on the photo with the pencil tool and use the **Color** and **Size** controls to select the type of pencil to use. When you select the pencil button the cursor changes to a pencil and you can then draw on the photo, as shown in Figure 9.

Figure 9 Drawing on a photo with the pencil tool

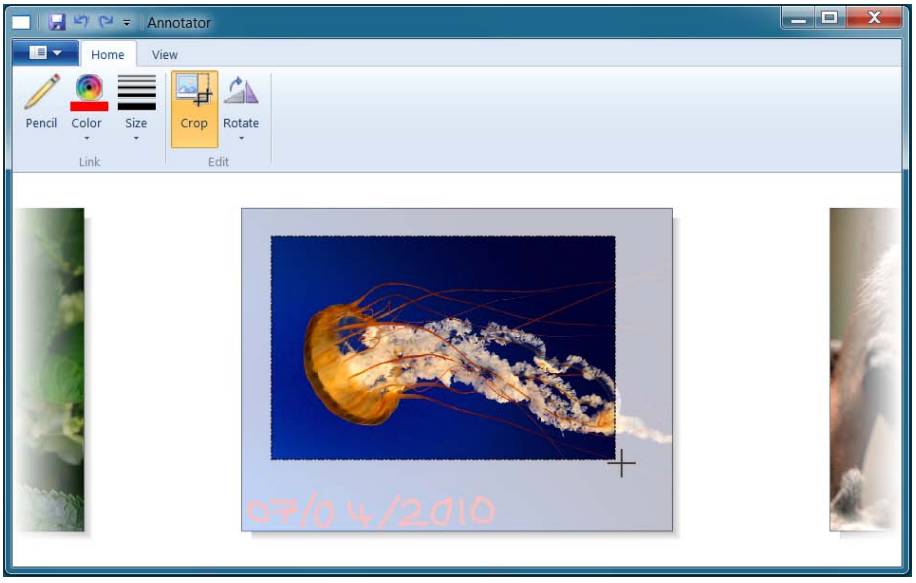


When you have changed a photo, you'll see that the **Undo** button is enabled (the third icon from the left on the title bar in Figure 9). Annotator keeps a list of every change that you make to a photo and the **Undo** button allows you to undo one of these steps (the **Redo** button allows you to redo a step that you have undone).

When the **Pencil** button is selected the cursor becomes a pencil and when you use the stylus on a touch screen or move the mouse with the left button down the effect is to draw on the photo. If the **Pencil** button is deselected the cursor changes to an arrow and when you use the stylus or the mouse with the left button down the effect is to move the photo.This is useful if you have zoomed in so that the editor only shows part of the photo and you want to move to a different part.

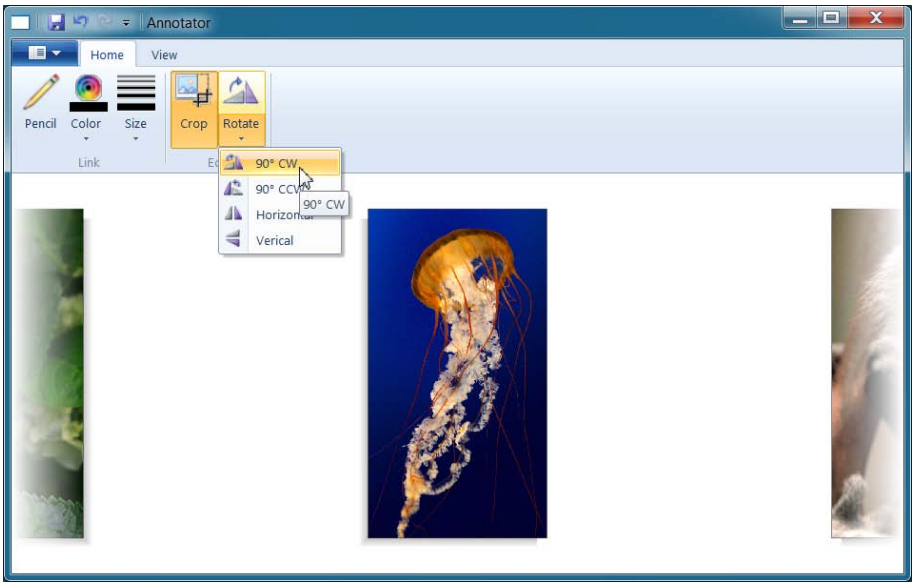
In some cases you'll want to crop a photo and to do this you use the crop tool. When you click the **Crop** button the cursor changes to cross-hairs and the entire photo is grayed out. You can now use the stylus or the mouse to draw the new boundaries of the photo, Figure 10.

Figure 10 Cropping a photo



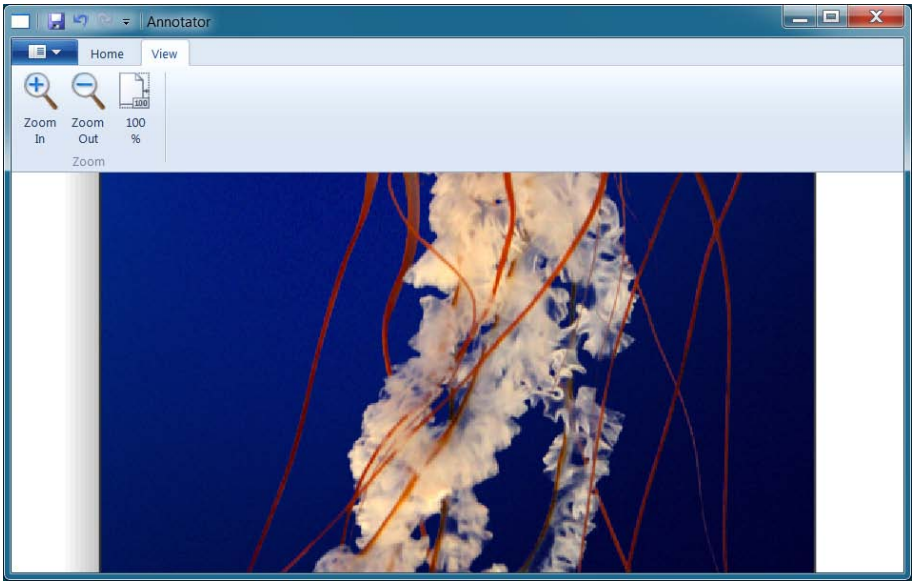
The Annotator also allows you to rotate and mirror the photo. You do this through the items on the **Rotate** drop down menu. Figure 11, shows a rotation of 90° clockwise. When you perform one of these operations Annotator animates the operation so that you see the rotation or mirror occurring rather than simply the final result. In addition, the image is zoomed so that it completely fills the space available, so in Figure 11, since the cropped image is taller than it is wide after the rotation, it is zoomed out (and appears smaller) so that the height of the photo fits the height of the editor pane. If you rotate the photo by another 90° the image will now be wider than it is high so it will be zoomed so that the new height fills the height of the editor.

Figure 11 Rotating a photo



if you want to examine the details of a photo, there are various ways to zoom in or out. First you can hold down the CTRL key and use the mouse wheel, second you can use the Plus Sign and Minus Sign keys and third, you can use the **Zoom In** and **Zoom Out** buttons on the **View** tab shown in Figure 12. To zoom to 100% you can press the ESC key or click the **100%** button.

Figure 12 Zooming a photo



When you exit the Annotator any changes that you have made are saved automatically, but you may also save the changes at any time by using the **Save** or **Save A Copy As** menu items. When you save a photo the Annotator makes a backup copy of the original photo in a subfolder called **AnnotatorBackup**.

Conclusion

The Hilo Annotator is the second application in the Hilo suite of applications and is used to edit photos in various ways. The Annotator provides tools to draw, crop, and rotate a photo and access to these tools is given through an instance of the Windows Ribbon control. Programming the Windows Ribbon control is the subject of the next chapter in the series.

Chapter 10: Using the Windows Ribbon

The Windows Ribbon control is designed to help users find, use, and understand available commands for a particular application in a way that's more natural and intuitive than menu bars or toolbars. Microsoft Office applications have used the Ribbon control since Office 2007 and Windows 7 applications, such as Paint, use the Ribbon control. The Ribbon control also provides benefits to the developer by separating presentation and logic. The Hilo Annotator application provides access to all of its tools through the Ribbon control. This article describes how the Annotator Ribbon is implemented.

Introducing the Ribbon Framework

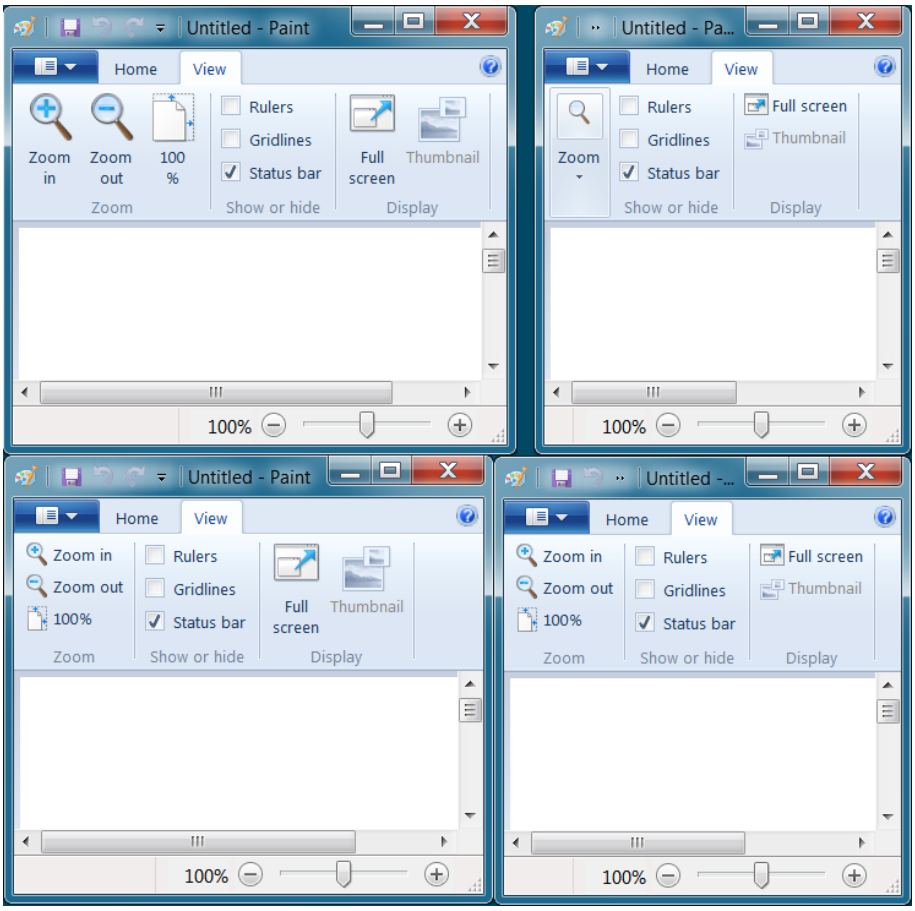
The Windows Ribbon control is a COM control and since it has a user interface you must initialize an STA (single threaded) apartment. The Windows Ribbon control is not an ActiveX control. This means that you do not have to provide an ActiveX control site, which simplifies considerably the code that you have to write in your application.

The Ribbon control uses adaptive layout. This means that the developer provides information about the controls that will be used and how they will be grouped, and at run time the Ribbon control determines the actual position of the controls.

To see the effect of adaptive layout you can run Windows 7 Paint and resize the window. This is shown in Figure 1. The top left image shows the Ribbon control with the **View** tab selected. At this width the items on the Ribbon control are shown full size. When the window width is reduced, the Ribbon control width is reduced and adaptive layout resizes the controls to enable all of them to be shown.

The bottom left image in Figure 1 shows the first change, that the **Zoom** group has compacted from a row of three buttons to a column of buttons. When the width is reduced further (bottom right, Figure 1) the **Display** group collapses to a column of buttons. At this size, there is no space to show the Customize Quick Access Toolbar button on the title bar, so instead there is a single button labeled **..** and when you click on this button the toolbar pops up. The most compact arrangement (top right, Figure 1) collapses the **Zoom** group to a drop-down menu. If the window width is reduced further, the items on the Ribbon control cannot be shown and it disappears completely.

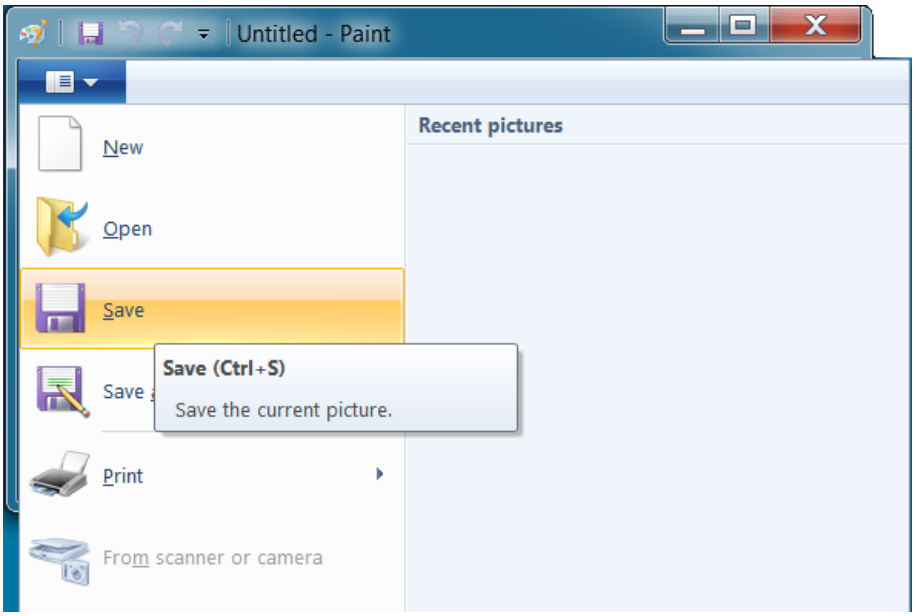
Figure 1 Adaptive layout, counter-clockwise, occurring as the window width is reduced



Adaptive layout is a consequence of the separation of presentation and logic. You design the user interface (UI) with a design tool that generates XAML, compile this to a BML file, and bind it to the application as a **UIFILE** resource. You do not have to write layout code, resizing code, child control creation, or initialization code. All of this is provided by the Ribbon control based on the markup information in the BML file, which is passed to the control when it is first initialized. The commands on the Ribbon control generate command messages, so you have to write code to handle these command messages. To do this you create a COM object called a command handler object.

The XAML for the presentation is made up of two sections. One section defines the command names including a name for use in the XAML and a unique ID used to identify the command in the code. The command section also allows you to define command specific properties like a label, a tooltip, or image. The other section provides information about the controls that generate the command messages. More than one control can generate a command, and the practice of defining the commands separate from the controls means that all controls that generate a command will have the same label, tooltip, image, and so on. This is illustrated in Figure 2 where you can see that the **Save** menu item displays a tooltip, a label, and a large icon. The **Save** item on the Quick Access Toolbar displays a small icon and no label on the toolbar, however, the toolbar shows the same tooltip as the menu item and the same label is used on the quick access toolbar customize menu, indicating that the two items are associated with the same command.

Figure 2 Illustrating command properties shown by Ribbon controls



Controls [http://msdn.microsoft.com/en-us/library/dd940497(v=VS.85).aspx] are not created on their own, instead they are hosted in a container called a view and they may be grouped together. The Ribbon control API supports two types of view: Ribbon View [http://msdn.microsoft.com/en-us/library/dd316811(v=VS.85).aspx] and ContextPopup View [http://msdn.microsoft.com/en-us/library/dd371654(v=VS.85).aspx] . The Ribbon View contains the application menu, tabs, and the Quick Access Toolbar; the ContextPopup View supports context menus and mini toolbars. These containers can have individual controls or have groups of controls. Grouping together controls helps the user by categorizing controls that perform similar tasks, and it helps the Ribbon control adaptive layout as shown in Figure 1.

Each control will have properties that can be accessed at runtime and define the behavior of the control. Some of these properties can be set in the markup XAML code. For example the **Button** [http://msdn.microsoft.com/en-us/library/dd940490(v=VS.85).aspx] control has properties for the label, tooltip, and icon that are provided by the command associated with the button. The Ribbon control framework provides access to property values through code.

Adding A Ribbon Resource

A Windows Ribbon control must be initialized with presentation information provided by a BML resource bound to the executable. The BML resource is compiled from markup code provided as XAML.

Writing the Markup File

The first step in creating the Ribbon control presentation is to write the XAML code. Listing 1 shows the basic format of the source XAML file. The **<Application>** [http://msdn.microsoft.com/en-us/library/dd371599(v=VS.85).aspx] element has two child elements, **<Application.Commands>** [http://msdn.microsoft.com/en-us/library/dd371598(v=VS.85).aspx] and **<Application.Views>** [http://msdn.microsoft.com/en-us/library/dd371600(v=VS.85).aspx] , the names of these elements indicate that they are XAML property elements [http://msdn.microsoft.com/en-us/library/ms788723.aspx#property_element_syntax] which means that these child elements are actually treated as properties of the **<Application>** element, rather than child elements. Any attribute of a XAML element can be provided by using property elements, but they are usually used to provide complex objects to a property, and in this case **<Application.Commands>** provides a collection of **<Command>**elements [http://msdn.microsoft.com/en-us/library/dd371595(v=VS.85).aspx] and **<Application.Views>** provides a collection of one or more of the view elements [http://msdn.microsoft.com/en-us/library/dd371597(v=VS.85).aspx] .

Listing 1 Basic XAML for a Ribbon control

XAML

```
<Application xmlns="http://schemas.microsoft.com/windows/2009/Ribbon">
```

```
    <Application.Commands>
    </Application.Commands>
    <Application.Views>
    </Application.Views>
</Application>
```

To be useful each command element must have a name and symbol. The name is a string used by other XAML code to refer to the command, and the symbol is an identifier that code will use to identify the command. For example, Listing 2 is an extract from the markup code for Hilo Annotator. This markup describes the command that is used by the **Open** application menu item. The **Name** attribute gives the string, **openFileMenu**, that is used to associate the command with the menu item control. The **Symbol** attribute gives the name of an integer symbol that the XAML compiler will create in a header file and your code will use this to identify the command. If you do not provide a **Symbol** item then the XAML compiler will generate a symbol for you. Listing 2 shows several property elements, and these provide string values. Property elements are used rather than XML attributes because the element is used to provide an **Id** value so that the string will be placed in a string table resource as part of the executable.

Listing 2 Defining a command

XAML

```
<Command Name="openFileMenu" Symbol="ID_FILE_OPEN" Comment="Open">
   <Command.LabelTitle>
      <String Id ="520">Open</String>
   </Command.LabelTitle>
   <Command.LargeImages>
      <Image Id ="521">res/open_32.bmp</Image>
   </Command.LargeImages>
   <Command.SmallImages>
      <Image Id="522">res/open_16.bmp</Image>
   </Command.SmallImages>
</Command>
```

The **<Application.Views>** element is mandatory and you use this to provide one or more of the view elements. Listing 3 shows an extract from the markup for Hilo Annotator. Annotator only provides the **<Ribbon>** [http://msdn.microsoft.com/en-us/library/dd316811(v=VS.85).aspx] element for the **<Application.Views>** property. The Ribbon control provides definitions for the main application menu through the **<Ribbon.ApplicationMenu>** [http://msdn.microsoft.com/en-us/library/dd316796(v=VS.85).aspx] property element, a Quick Access Toolbar provided through the **<Ribbon.QuickAccessToolbar>** [http://msdn.microsoft.com/en-us/library/dd316816(v=VS.85).aspx] element, and information about the Ribbon control tabs through the **<Ribbon.Tabs>** [http://msdn.microsoft.com/en-us/library/dd316826(v=VS.85).aspx] element.

Listing 3 Illustrating the Views property

XAML

```
<Ribbon>
   <Ribbon.ApplicationMenu>
      <ApplicationMenu CommandName="fileMenu">
         <MenuGroup>
            <Button CommandName="openFileMenu" />
            <Button CommandName="saveFileMenu" />
            <Button CommandName="saveAsFileMenu" />
         </MenuGroup>
         <MenuGroup>
            <Button CommandName="exitMenu" />
         </MenuGroup>
      </ApplicationMenu>
   </Ribbon.ApplicationMenu>
   <Ribbon.Tabs>
      <!-- code omitted -->
```

```
    </Ribbon.Tabs>
    <Ribbon.QuickAccessToolbar >
       <!-- code omitted -->
    </Ribbon.QuickAccessToolbar >
</Ribbon>
```

As you can see from Listing 3, the **openFileMenu** command defined in Listing 2 is used as one menu item on the application menu. The markup shows two unnamed **<MenuGroup>** [http://msdn.microsoft.com/en-us/library/dd371700(v=VS.85).aspx] elements and in practice this will mean that the child items in these groups will be shown in the menu as if they are a single group. However, if you provide a **CommandName** property then the Ribbon control will display the string provided through the command object's **LabelTitle** property as an annotation to the menu items

Describing the Ribbon Tabs with XAML

The most noticeable part of the Windows Ribbon control are the tabs showing controls and these are described by using the **<Ribbon.Tabs>** [http://msdn.microsoft.com/en-us/library/dd316826(v=VS.85).aspx] property element. This element contains one or more **<Tab>** [http://msdn.microsoft.com/en-us/library/dd316894(v=VS.85).aspx] elements. Hilo Annotator defines two tabs, **Home** and **View**, Listing 4 shows an excerpt from the markup that defines the **View** tab. This tab has a single group called **Zoom**, which has three buttons and the tab uses a scaling policy to indicate how the buttons will be arranged when the Ribbon control is resized.

Listing 4 Defining the Ribbon tabs

XAML

```
<Ribbon.Tabs>
   <Tab CommandName="homeTab">
      <!-- code omitted -->
   </Tab>
   <Tab CommandName="viewTab">
      <Tab.ScalingPolicy>
         <ScalingPolicy>
            <!-- The following list the maximum sizes of the groups -->
            <ScalingPolicy.IdealSizes>
               <Scale Group="zoomGroup" Size="Large" />
            </ScalingPolicy.IdealSizes>
            <!-- The following items give the shrink order of the groups -->
            <Scale Group="zoomGroup" Size="Medium" />
         </ScalingPolicy>
      </Tab.ScalingPolicy>
      <Group CommandName="zoomGroup" SizeDefinition="ThreeButtons">
         <Button CommandName="zoomInButton" />
         <Button CommandName="zoomOutButton" />
         <Button CommandName="previewButton" />
      </Group>
   </Tab>
</Ribbon.Tabs>
```

The **<ScalingPolicy>** [http://msdn.microsoft.com/en-us/library/dd316839(v=VS.85).aspx] element has two purposes: specifying the ideal size of each group and specifying what order the groups collapse during adaptive layout as the Ribbon control size decreases. The **<ScalingPolicy.IdealSizes>** [http://msdn.microsoft.com/en-us/library/dd316842(v=VS.85).aspx] element has a **<Scale>** element for each group on the tab and the **Size** attribute gives the ideal size of the group. There are four possible values for the **Size** attribute: **Small**, **Medium**, **Large** and **Popup**. However, the value that you can use depends on the size definition template used by the group, provided by the value of the **SizeDefinition** [http://msdn.microsoft.com/en-us/library/dd316927(v=VS.85).aspx] attribute. For example, the **zoomGroup** has a **SizeDefinition** value of **ThreeButtons** and this defines two layouts: three buttons in a row or three buttons in a column. These two layouts correspond to the **Size** attribute values of **Large** and **Medium** respectively. The **<Scale>** elements outside of the **<ScalingPolicy.IdealSize>** element determine the order that groups collapse to a smaller size. Since the **Zoom** tab only has one group there is just one element. If there were more groups then the order of the **<Scale>** items would determine the order of the group shrinkage.

The Ribbon control supports many different controls. Some are simple like **Button** and **Check Box** [http://msdn.microsoft.com/en-us/library/dd940491(v=VS.85).aspx] , but the Ribbon control also provides more complex controls through elements called galleries. Hilo Annotator uses a **DropDownGallery** control to provide the drop-down list of the pencil widths. The markup code for this control is very simple, as shown in Listing 5. This markup simply indicates that the control is shown to the user as a button and the **sizeButton** command gives the images that will be shown on the button.

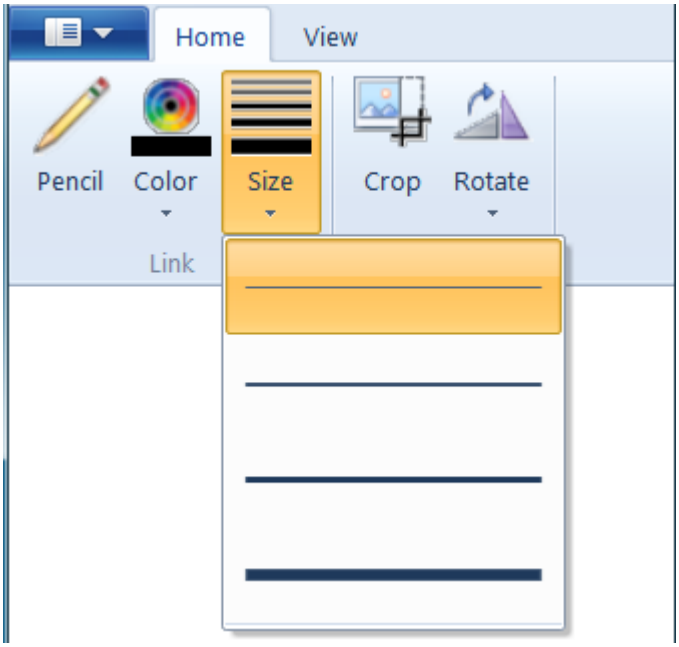
Listing 5 Declaring the markup code for Pencil size control

XAML

```
<DropDownGallery CommandName="sizeButton" TextPosition="Hide"
   Type="Items" ItemHeight="32" ItemWidth="128" HasLargeItems="true">
   <DropDownGallery.MenuLayout>
      <VerticalMenuLayout Gripper="None" />
   </DropDownGallery.MenuLayout>
</DropDownGallery>
```

When you click on a drop-down gallery button the gallery list is shown. The **Gripper** property has a value of **None** which indicates that the user cannot resize the gallery list since there is no gripper resizing handle. Figure 3 shows the drop-down gallery displayed when the user clicks the **Size** button. There is no information about the drop-down list in the markup in Listing 5 because this information is obtained at run time. The items in a **DropDownGallery** control are accessed through the **IUICollection** [http://msdn.microsoft.com/en-us/library/dd371519(v=VS.85).aspx] interface implemented by the control, during the initialization of the Ribbon control the application's command handler object accesses the **IUICollection** interface and uses it to add the images.

Figure 3 Displaying the size gallery



Binding the BML Resource to the Executable

The only way to initialize a Ribbon control with the markup information is to call the **IUIFramework::LoadUI** [http://msdn.microsoft.com/en-us/library/dd371471(v=VS.85).aspx] method and pass the name of a **UIFILE** resource bound to the executable. To do this you must compile the XAML to produce a BML file and bind this to the executable. The tool to compile the XAML to BML is called uicc.exe and is provided as part of the Windows 7 Software Development Kit (SDK). This tool produces three outputs: the BML file that contains the markup information, a header file that contains the symbols that identify the commands, and a resource script that describes the string tables, bitmaps and the **UIFILE** resource for the BML file. The following steps describe how to add the XAML markup file to a project and add a build step to use the uicc.exe tool. These steps are for Visual C++ Express 2010 and all versions of Visual Studio 2010.

To add the XAML markup file and build step to a project:

1. Add the XML file to the project using Solution Explorer.
2. In Solution Explorer, right-click the XML file, and then click the **Properties** item to show the property page. On the **General** page, change the **Item Type** item to **Custom Build Tool**, and click the **Accept** button. This will ensure that the **Custom Build Tool** page is shown on the property page.
3. In the **Configuration** drop-down list, click **All Configurations**.
4. Click the **Custom Build Tool** page, and in the **Command Line** item type the following

   uicc.exe "%(FullPath)" %(Filename).bml /header:%(Filename).h /res: %(Filename).rc

   This indicates that the uicc.exe tool is used to compile the XML file and provides names for the BML file, the header file, and resource script.

5. Click **OK,** to save the values and close the property pages.
6. Now add a line to insert the generated resource script to the project's resource script. In Solution Explorer right-click the project's resource script file (for example in Hilo Annotator this is called **Annotator.rc**) and click **View Code**.
7. Scroll to the bottom of the file and add a line like the following where **RibbonRes.rc** is the name of the resource script created by uicc.exe

   #include "RibbonRes.rc"

8. Press **Ctrl+S** to save the resource script.

After these steps when you build the project the XML file will be compiled and the resource bound to the executable.

When you build the solution, if you get an error that uicc.exe is not a command (this will happen if you run Visual C++ 2010, a 32-bit application, on an Windows 7 x64-based system) then it means that you need to add the path to the SDK tools to the project's properties.

To add the path to the SDK tools to the project properties:

1. In Solution Explorer right-click the project, and then click **Properties**.
2. In the **Configuration** category, click **VC++ Directories**.
3. Edit the **Executable Directories** item to add the path to the **Bin** folder of the Windows 7 SDK.

Using the Ribbon Control

To use the Ribbon control in your application you must create it through COM, and initialize the user interface with the BML file and provide a command handler object to handle the command messages generated when the user interacts with the controls on the Ribbon control. The header for the Ribbon control is uiribbon.h. This file defines the interfaces, control property keys, and the Class Id (CLSID) needed to create and use the Ribbon control.

Initializing the Ribbon

The Ribbon control has to be activated with a call to **CoCreateInstance** and in Hilo Annotator this occurs in the **AnnotatorApplication::InitializeRibbonFramework** method, which is called when the Annotator window is first created. The relevant lines are shown in Listing 6 which returns a pointer to the **IUIFramework** [http://msdn.microsoft.com/en-us/library/dd371467(v=VS.85).aspx] interface. This interface gives access to the core functionality of the Ribbon control.

Listing 6 initializing the Ribbon control

C++

```
HRESULT hr = CoCreateInstance(
    CLSID_UIRibbonFramework, NULL, CLSCTX_INPROC_SERVER,
    IID_PPV_ARGS(&m_ribbonFramework));

HWND hWnd = nullptr;
```

```
if (SUCCEEDED(hr))
{
   // window is the main application window
   hr = window->GetWindowHandle(&hWnd);
}

if (SUCCEEDED(hr))
{
   // Initilize the ribbon framework
   hr = m_ribbonFramework->Initialize(hWnd, this);
}

if (SUCCEEDED(hr))
{
   static const int MaxResStringLength = 100;
   wchar_t ribbonMarkup[MaxResStringLength] = {0};
   // Obtain the name of the BML resource
   ::LoadString(
      HINST_THISCOMPONENT, IDS_RIBBON_MARKUP,
      ribbonMarkup, MaxResStringLength);

   hr = m_ribbonFramework->LoadUI(GetModuleHandle(NULL), ribbonMarkup);
}
```

After creating the Ribbon object the code in Listing 6 obtains the handle for the main application window and passes this to the Ribbon control by calling the **IUIFramework::Initialize** [http://msdn.microsoft.com/en-us/library/dd371467(v=VS.85).aspx] method. The Ribbon control is not an ActiveX control so it does not need the application to implement control site interfaces. Instead the control is created as a child window of the window whose handle you pass as the first parameter. The second parameter, **Initialize** is a pointer to an **IUIApplication** [http://msdn.microsoft.com/en-us/library/dd371528(v=VS.85).aspx] interface implemented by the application the **this** pointer is passed because the **AnnotatorApplication** class implements this interface. Finally the **InitializeRibbonFramework** method calls **IUIFramework::LoadUI** [http://msdn.microsoft.com/en-us/library/dd371471(v=VS.85).aspx] to provide information about the presentation of the Ribbon control. This method will only load a BML file bound as a resource to an executable (an EXE or DLL), so you have to provide the instance handle of the executable and the name of the BML file that has been bound as a **UIFILE** resource. The default name of this resource is **APPLICATION_RIBBON** and since Hilo Annotator stores this in the application's string table the **InitializeRibbonFramework** method calls the **LoadString** [http://msdn.microsoft.com/en-us/library/ms647486(VS.85).aspx] Windows 7 API to load this string.

The **IUIApplication** [http://msdn.microsoft.com/en-us/library/dd371528(v=VS.85).aspx] interface is essentially the site interface of the application which the Ribbon control uses to initialize the handshake mechanism between the application and the control. There are three methods on this interface, the **IUIApplication::OnCreateUICommand** [http://msdn.microsoft.com/en-us/library/dd371531(v=VS.85).aspx] method is called for every command when the Ribbon control is first created, the **IUIApplication::OnDestroyUICommand** [http://msdn.microsoft.com/en-us/library/dd371534(v=VS.85).aspx] method is called for every command when the Ribbon control is destroyed, and the **IUIApplication::OnViewChanged** [http://msdn.microsoft.com/en-us/library/dd371537(v=VS.85).aspx] method is called when the state of one of the view objects changes.

In Hilo Annotator the implementation of **OnViewChanged** is used to obtain the height of the Ribbon control by calling a method called **GetRibbonHeight**, as shown in Listing 7. This code illustrates how you can obtain the various views on the Ribbon control (the Ribbon or a **ContextPopup** [http://msdn.microsoft.com/en-us/library/dd371654(v=VS.85).aspx] object).

Listing 7 Obtaining the Ribbon control height when the view changes

C++

```
unsigned int AnnotatorApplication::GetRibbonHeight()
{
   unsigned int ribbonHeight = 0;
```

```
    if (m_ribbonFramework)
    {
       ComPtr<IUIRibbon> ribbon;
       HRESULT hr = m_ribbonFramework->GetView(0, IID_PPV_ARGS(&ribbon));

       if (SUCCEEDED(hr))
       {
          hr = ribbon->GetHeight(&ribbonHeight);

          if (FAILED(hr))
          {
             ribbonHeight = 0;
          }
       }
    }

    return ribbonHeight;
 }
```

When the Ribbon control is created it calls the application's **OnCreateUICommand** method for each command mentioned in the BML file, requesting a command handler object to handle command messages from the command. Through this method, the Ribbon control passes the ID of the command as specified in the markup, and an ID that indicates the type of the command. The final parameter of this method is an out parameter which the application uses to return an interface pointer to the command handler object. Hilo Annotator has a class called **UICommandHandler** that implements the **IUICommandHandler** [http://msdn.microsoft.com/en-us/library/dd371491(v=VS.85).aspx] interface. Annotator creates just one instance of this class when it first starts and returns this as the command handler object for every command, as shown in Listing 8.

Listing 8 Returning the command handler object

C++

```
 HRESULT AnnotatorApplication::OnCreateUICommand(
    unsigned int, UI_COMMANDTYPE, IUICommandHandler** commandHandler)
 {
    // The ribbon uses only one command handler
    return m_commandHandler->QueryInterface(IID_PPV_ARGS(commandHandler));
 }
```

The majority of the communication between the application and the Ribbon control object occurs through the command object.

Providing the Command Object

The command object implements the **IUICommandHandler** interface, which has just two methods, **Execute** [http://msdn.microsoft.com/en-us/library/dd371489(v=VS.85).aspx] and **UpdateProperty** [http://msdn.microsoft.com/en-us/library/dd371494(v=VS.85).aspx] . When a user interacts with any control on the Ribbon control a command message is generated and the Ribbon control informs the application by calling the **IUICommandHandler::Execute** method shown in Listing 9.

Listing 9 The IUICommandHandler::Execute method

C++

```
 HRESULT Execute(
     UINT32 commandId,
     UI_EXECUTIONVERB verb,
     const PROPERTYKEY *key,
     const PROPVARIANT *currentValue,
     IUISimplePropertySet *commandExecutionProperties
 );
```

The first two parameters of this method are identifiers indicating the command and the action that has occurred.

The **commandId** parameter will be one of the symbols in the header file created by the uuic.exe tool and it will be either a symbol that you specified in the markup, or if you did not specify then the uicc.exe will create a symbol for you. The **verb** parameter indicates the action of the user and can indicate that the command was executed, or that the control was previewed (the mouse is hovered over a control), or a preview was cancelled. Clearly a value of **UI_EXECUTIONVERB_EXECUTE** [http://msdn.microsoft.com/en-us/library/dd371563(v=VS.85).aspx] , indicating that the command has executed, is the most important.

Commands can be quite complex since a control can be composed of several controls, each of which will have values. These values are the control properties. The **Execute** method indicates the property that is the subject of the command through the **key** parameter and the value of the property through **currentValue**. The property identifier is a pointer to a **PROPERTYKEY** [http://msdn.microsoft.com/en-us/library/bb773381(VS.85).aspx] structure, and the values in the structure identify the type of the property and a unique identifier. You do not need to know the values used, all you need to know is the symbol. The uiribbon.h header file defines the properties that are used by the Ribbon control, so for example, the **UI_PKEY_ENABLED** symbol is used to identify the enabled property and indicates that this property is a **Boolean**.

The actual value of the property is accessed through the **currentValue** parameter which is a pointer to a **VARIANT**. The final parameter of the **Execute** method is a pointer to a **IUISimplePropertySet** [http://msdn.microsoft.com/en-us/library/dd371358(v=VS.85).aspx] interface that you can use to access other properties of the command.

Hilo Annotator provides an implementation of **IUICommandHandler** interface through the class **UICommandHandler**. The implementation in Annotator, **UICommandHandler::Execute**, has a large **switch** statement so that it can provide different code for each command and much of the code delegates the code to other objects in the application. For example, Listing 10 shows part of this **switch** statement, where the **m_imageEditor** variable is a pointer to the object that provides the code for the image editor child window in the lower half of the Annotator window. You will recognise the **ID_FILE_OPEN** symbol from Listing 2 where it is declared as the symbol for the **openFileMenu** command.

Listing 10 Handling command messages

C++

```
HRESULT hr = S_OK;

switch (commandId)
{
case ID_FILE_OPEN:
   {
      hr = m_imageEditor->OpenFile();
      break;
   }
case ID_FILE_SAVE:
   {
      m_imageEditor->SaveFiles();
      break;
   }
case ID_FILE_SAVE_AS:
   {
      m_imageEditor->SaveFileAs();
      break;
   }
case ID_FILE_EXIT:
   {
      m_imageEditor->SaveFiles();
      ::PostQuitMessage(0);
      break;
   }
// other code omitted
}
```

Accessing Command Properties

The controls [http://msdn.microsoft.com/en-us/library/dd940497(v=VS.85).aspx] that you can use on a Ribbon control are documented in the MSDN Library Each page lists the property keys that the control supports. The documentation describes how some of the properties are accessed through calls to the Ribbon control's **IUIFramework** interface (the **IUIFramework::GetUICommandProperty** [http://msdn.microsoft.com/en-us/library/dd371370(v=VS.85).aspx] and **IUIFramework::SetUICommandProperty** [http://msdn.microsoft.com/en-us/library/dd371478(v=VS.85).aspx] methods). However, most of the properties are accessible through a process that the documentation calls invalidation.

Invalidation means that the application tells the Ribbon control that a control property is invalid and then the Ribbon control calls the application to request the new value. In Hilo Annotator invalidation occurs in the image editor code whenever the Image Editor window is redrawn or if a command is executed. Listing 11 shows this code where the **m_framework** variable is the **IUIFramework** interface on the Ribbon control.

Listing 11 Invalidating control properties

C++

```
HRESULT ImageEditorHandler::UpdateUIFramework()
{
   // After we're done drawing make sure to update framework buttons as necessary
   if (m_framework)
   {
      m_framework->InvalidateUICommand(
         ID_BUTTON_UNDO, UI_INVALIDATIONS_STATE, nullptr);
      m_framework->InvalidateUICommand(
         ID_BUTTON_REDO, UI_INVALIDATIONS_STATE, nullptr);
      m_framework->InvalidateUICommand(
         ID_FILE_SAVE, UI_INVALIDATIONS_STATE, nullptr);

      m_framework->InvalidateUICommand(
         pencilButton, UI_INVALIDATIONS_VALUE, nullptr);
      m_framework->InvalidateUICommand(
         cropButton, UI_INVALIDATIONS_VALUE, nullptr);
    }

    return S_OK;
}
```

The **IUIFramework::InvalidateUICommand** [http://msdn.microsoft.com/en-us/library/dd371375(v=VS.85).aspx] method can be used to invalidate just one property of a command, or all properties of a particular type. The first parameter of this method is an identifier indicating the command to update, the second parameter indicates the type of data to update (a **UI_INVALIDATIONS** [http://msdn.microsoft.com/en-us/library/dd371573(v=VS.85).aspx] value), and the final parameter is the ID of a property if you wish to update a specific property. In Listing 11 the state is invalidated for the **Undo**, **Redo**, and **Save** buttons so that the Ribbon control requests that the application returns the enabled state (**UI_PKEY_Enabled**) of the **Undo** and **Save** buttons.

The **pencilButton** and **cropButton** controls are **Toggle Button** [http://msdn.microsoft.com/en-us/library/dd940509(v=VS.85).aspx] controls and in Hilo Annotator only one of them can be pushed in. This pushed-in state is a Boolean value so rather than requesting the change in property state, the call to the **InvalidateUICommand** method for these two controls requests that the values of the control properties are invalidated, and the application will check for, and return a value for the **UI_PKEY_BooleanValue** of these controls.

Once you have invalidated a command, the Ribbon control cannot display the associated control's state, so it must ask the application for the property values by calling the **IUICommandHandler::UpdateProperty** [http://msdn.microsoft.com/en-us/library/dd371494(v=VS.85).aspx] method. This method is called with four parameters, the first three indicate the command, the property, and the current value of the property.The fourth is an out parameter for the new value of the property. In Hilo Annotator this method is implemented in the **UICommandHandler** class and has two purposes. The first is to return the state and value properties invalidated by the **UpdateUIFramework** method (Listing 11)1 The second purpose is to populate the **DropDownGallery**

control for the pencil widths with appropriate images.

Listing 5 shows that the markup for the Gallery control does not provide the values to be shown by the control. When the control is initialized, the Ribbon control calls the **UpdateProperty** method to update the **UI_PKEY_ItemsSource** [http://msdn.microsoft.com/en-us/library/dd371348(v=VS.85).aspx] property. The **UICommandHandler** class implementation of this method accesses the **IUICollection** [http://msdn.microsoft.com/en-us/library/dd371519(v=VS.85).aspx] interface on this property and adds the images to be shown by the Gallery control.

Conclusion

In this chapter you learned how Hilo Annotator uses a Windows Ribbon control to give easy access to the image editor tools. In the next article you will learn how to use the Windows 7 Imaging Component to load and convert images.

Chapter 11: Using the Windows Imaging Component

The Windows 7 Imaging Component (WIC) allows you to load and manipulate images and their metadata. The WIC Application Programming Interface (API) has built-in component support for all standard formats. In addition, the images created by the WIC can be used to create Direct2D bitmaps so you can use Direct2D to change images. In this article you will learn how the Windows 7 Imaging Component is used in the Hilo Browser and Annotator applications.

Introducing the WIC

Both the Hilo Browser and Annotator display photos, and Annotator allows you to alter photos. For Hilo the definition of a photo is any image type so the Hilo applications have to be able to load and display a wide range of file types. The Windows 7 Imaging Component provides this functionality. The WIC can load images that are made up of multiple frames and it can access metadata in the image file. The WIC supports all the common image formats and even allows developers to develop codecs (coder-decoder components) for new formats.

To use the WIC you must include the **wincodec.h** header file, it contains the definitions for the various interfaces used by the WIC, definitions of structures and GUIDs for the WIC objects, and the standard pixel formats. The WIC is not just one component, instead there are several components used to encode and decode the different formats supported. Different image formats store image data in different ways, so to load an image file you need a decoder component to decode the data into a format your application can use. When you save image information you need an encoder component to encode the data into the format defined by the image file format. If you know the type of image you wish to load then you can create a decoder with a call to **CoCreateInstanceEx** [http://msdn.microsoft.com/en-us/library/ms686615(VS.85).aspx] and provide the Class ID (CLSID) of the decoder object. If you do not know the image type you can use the WIC API to examine the file and choose the appropriate object. To do this you create an instance of the WIC factory object.

Listing 1 shows the **Direct2DUtility::GetWICFactory** method that is used by the Hilo applications to create an instance of the factory object. Like all the other WIC objects, the factory object is a COM object and so this code must be called in a COM apartment. You can initialize either an STA or an MTA apartment.

Listing 1 Creating a WIC factory

C++

```
HRESULT Direct2DUtility::GetWICFactory(IWICImagingFactory** factory)
{
    static ComPtr<IWICImagingFactory> m_pWICFactory;
    HRESULT hr = S_OK;

    if (nullptr == m_pWICFactory)
    {
        hr = CoCreateInstance(
              CLSID_WICImagingFactory, nullptr,
              CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&m_pWICFactory) );
    }

    if (SUCCEEDED(hr))
    {
        hr = AssignToOutputPointer(factory, m_pWICFactory);
    }

    return hr;
}
```

The **IWICImagingFactory** [http://msdn.microsoft.com/en-us/library/ee690281(v=VS.85).aspx] interface has

methods that allow you to create decoders and encoders, bitmaps and streams; and it allows you to create objects to alter color palettes and metadata in an image file.

Handling Images

Conceptually bitmap graphics can be viewed as rows of pixels with the color of each pixel composed of color components such as red, blue, and green or cyan, magenta ,and yellow, and an alpha channel component to determine opacity. If these components have 256 values, involving 32-bits per pixel, this would mean that a 1000 by 1000 pixel image would take up a megabyte of storage. Since images of this resolution are fairly standard, and storage space is always at a premium, there is a need to compress the way that pixels are represented to reduce image size. In most images there are repetitions of pixels, and most images do not use the full gamut of colors, and so image compression techniques have been developed to reduce this redundancy. Some techniques are lossless, so that all the original pixels can be recreated during decompression and others are lossy meaning that to the human eye the decompressed image looks like the original but does not necessarily have the same pixels.

A codec is code that **co**mpresses and **dec**ompresses images, and code that compresses an image is called an encoder and the code that decompresses images is called a decoder. Your graphics card will have its own image format and so a codec must be able to convert the image format used by an image file to and from a form that Windows 7 can use. The WIC decoders can provide raw pixel information that can be used with GDI functions. In addition, Direct2D can create a bitmap object (with an **ID2D1Bitmap** [http://msdn.microsoft.com/en-us/library/dd371109(v=VS.85).aspx] interface) from a WIC image so you can use Direct2D to draw an image loaded by WIC. You can also create a Direct2D render target based on a WIC image, which means that you can use Direct2D drawing actions to change the WIC image.

Images can be provided in several formats. Typically an image will be a file on a hard disk so WIC provides methods to load an image from a file name or a file handle. Images may also be stored as streams within other files (OLE documents) so WIC provides methods to load from an OLE stream (the **IStream** [http://msdn.microsoft.com/en-us/library/aa380034(VS.85).aspx] interface). Images may be stored as resources as part of executables and the WIC API provide methods to load images through **HBITMAP** or **HICON** data.

In addition to pixel information most image formats contain metadata. Digital cameras use metadata to store information such as the camera settings (ISO, shutter speed, aperture, flash) and information about the photo (time, date, GPS reading). There are several standard metadata formats: Exif, XMP, IPTC, are three of the common formats supported by WIC. These formats determine how the metadata are stored in an image file and also define standard metadata items. It is clearly the responsibility of a codec to handle the metadata formats used by the image type.

Some image formats support storing more than one image in a file, for example GIF images and some TIFF images. Many digital cameras also provide a thumbnail image as well as the full size image. WIC provides methods to get each of the images in a file as a frame, and it supports loading global thumbnails and frame-based thumbnails. The standard decoders do not support accessing global image data, so even if the image file only contains one image you still have to load that as a single frame and then use a frame decoder to get information about that frame.

Using Bitmap Sources

The WIC API provides a polymorphic collection of objects to give access to bitmap images. The base interface of these objects is **IWICBitmapSource** [http://msdn.microsoft.com/en-us/library/ee690171(v=VS.85).aspx] , which provides the pixel format, the size and resolution of the image, and allows you to make a copy of the pixels. The other interfaces are for objects that add additional functionality, for example to resize the image, to change the pixel format, or rotate the image. These interfaces are shown in Figure 1.

Figure 1 WIC Bitmap interfaces

The interface hierarchy means that wherever an **IWICBitmapSource** interface is required an instance of any of the other interfaces can be used. This means that you can pass an **IWICBitmapScaler** [http://msdn.microsoft.com/en-us/library/ee690168(v=VS.85).aspx] interface, implemented by a scaler object, to a function that requires an **IWICBitmapSource.** As the function reads information about the bitmap through the interface, the scaler object will perform scaling automatically on the information. Most of the child interfaces shown in Figure 1 simply add an **Initialize** method to the **IWICBitmapSource** interface to enable you to provide

information about how the object works, for example for the **IWICBitmapScalar** interface the **Initialize** [http://msdn.microsoft.com/en-us/library/ee690169(v=VS.85).aspx] method provides the new height and width to which the image will be scaled.

Loading Images with the WIC

To load an image you have to use a decoder object that implements the **IWICBitmapDecoder** [print-Hilo-2011_//msdn.microsoft.com/en-us/library/ee690086(v=VS.85).aspx] interface. The decoder object will decode only a specific image format and before you can use the decoder you must provide it with a stream that contains the image of the correct format. To do this you call the **IWICBitmapDecoder::Initialize** [http://msdn.microsoft.com/en-us/library/ee690108(v=VS.85).aspx] method. Once the decoder has been initialized you can call the **IWICBitmapDecoder::GetFrameCount** [http://msdn.microsoft.com/en-us/library/ee690099(v=VS.85).aspx] method to obtain the number of frames in the image and you can call the **IWICBitmapDecoder::GetFrame** [http://msdn.microsoft.com/en-us/library/ee690098(v=VS.85).aspx] method with an index to get a specific decoded frame which is returned through an object with the **IWICBitmapFrameDecode** [http://msdn.microsoft.com/en-us/library/ee690134(v=VS.85).aspx] interface.

The **IWICBitmapFrameDecode** interface derives from the **IWICBitmapSource** interface and so has methods to obtain information about the image. For example, you can call the **IWICBitmapSource::GetPixelFormat** [http://msdn.microsoft.com/en-us/library/ee690181(v=VS.85).aspx] to get information about the pixel format (one of the GUIDs given on the Native Pixel Formats Overview [http://msdn.microsoft.com/en-us/library/ee719797(v=VS.85).aspx] page) and then obtain the actual pixels by calling the **IWICBitmapSource::CopyPixels** [http://msdn.microsoft.com/en-us/library/ee690179(v=VS.85).aspx] method. You can also call the **IWICBitmapFrameDecode::GetMetadataQueryReader** [http://msdn.microsoft.com/en-us/library/ee690137(v=VS.85).aspx] method to obtain a metadata query object with the **IWICMetadataQueryReader** [http://msdn.microsoft.com/en-us/library/ee719708(v=VS.85).aspx] interface through which you can obtain metadata about the frame.

Images can have different pixel formats varying by the number of bits used for each color component, the order of the color components, whether there is an alpha channel ,and whether a palette is used. Direct2D requires bitmap sources to be in the **32bppPBGRA** format which means that each pixel is 4 bytes with one byte each for blue, green, red, and the alpha channel, in that order. To render an image on a Direct2D render target you must convert an image to this format and make the image through an object with an **ID2D1Bitmap** [http://msdn.microsoft.com/en-us/library/dd371109(VS.85).aspx] interface.

Listing 2 Loading an image as a Direct2D bitmap

C++

```
ComPtr<IWICBitmapDecoder> decoder;
ComPtr<IWICBitmapFrameDecode> bitmapSource;
ComPtr<IWICFormatConverter> converter;

ComPtr<IWICImagingFactory> wicFactory;
GetWICFactory(&wicFactory);

wicFactory->CreateDecoderFromFilename(
   uri, // name of the file
   nullptr, GENERIC_READ, WICDecodeMetadataCacheOnLoad, &decoder);

decoder->GetFrame(0, &bitmapSource);
wicFactory->CreateFormatConverter(&converter);
converter->Initialize(
   bitmapSource, GUID_WICPixelFormat32bppPBGRA,
   WICBitmapDitherTypeNone, nullptr, 0.f,
   WICBitmapPaletteTypeMedianCut);
// Create a Direct2D bitmap from the WIC bitmap.
hr = renderTarget->CreateBitmapFromWicBitmap(converter, nullptr, bitmap);
```

Listing 2 is code taken from the **Direct2DUtility::LoadBitmapFromFile** method in the Hilo Common project. This method is used by the Browser and Annotator applications to load image files. First the code obtains the WIC factory object and then it loads the decoder for the image by calling the

IWICImagingFactory::CreateDecoderFromFilename [http://msdn.microsoft.com/en-us/library/ee690307(v=VS.85).aspx] method where the **uri** variable is the path to the image file. This method activates the appropriate decoder for the image type, loads the image file, and initializes the decoder with the image, so you do not need to call the **Initialize** method on the **IWICBitmapDecoder** interface returned from this method. The code in Listing 2 then calls **GetFrame** for the first frame in the bitmap to get the decoded image. The **IWICBitmapFrameDecode** interface uses the decoder object whenever information is read from the image.

The pixel format of this image is unlikely to be the 32bppPBGRA format so Listing 2 creates a converter object. The **IWICFormatConverter** [http://msdn.microsoft.com/en-us/library/ee690274(v=VS.85).aspx] interface derives from **IWICBitmapSource** and in essence, it delegates calls to this interface to the raw image, performing conversions where necessary. Finally, the code calls the **ID2D1RenderTarget::CreateBitmapFromWicBitmap** [http://msdn.microsoft.com/en-us/library/dd371797(v=VS.85).aspx] method to provide an object with the **ID2D1Bitmap** interface using data from the converted **IWICBitmapSource** object. The returned variable (in Listing 2 this is the **bitmap** variable) can be rendered on an **ID2D1RenderTarget** object and a Direct2D render target can be created from this bitmap to allow you to change the image using the Direct2D drawing operations. When you call the Direct2D bitmap object, the call is passed to the format converter, which calls the frame decoder, which calls the decoder object. At each stage the data is converted.

Manipulating Images with the WIC

Listing 2 shows that you can create a WIC object to change the format of the bitmap data that is supplied by an image, you can use the same technique to alter other aspects of the image. For example, you can use a scalar object to automatically resize an image. This is illustrated by the **Direct2DUtility::LoadBitmapFromFile** method. The code shown in Listing 2 loads the image without scaling, the code in Listing 3 shows the additional code to provide scaling.

Listing 3 Providing scaling

C++

```
ComPtr<IWICBitmapScaler> scaler;

if (destinationWidth != 0 || destinationHeight != 0)
{
   unsigned int originalWidth, originalHeight;
   bitmapSource->GetSize(&originalWidth, &originalHeight);
   if (destinationWidth == 0)
   {
      float scalar = static_cast<float>(destinationHeight) /
static_cast<float>(originalHeight);
      destinationWidth = static_cast<unsigned int>(scalar * static_cast<float>(originalWidth));
   }
   else if (destinationHeight == 0)
   {
      float scalar = static_cast<float>(destinationWidth) / static_cast<float>(originalWidth);
      destinationHeight = static_cast<unsigned int>(scalar *
static_cast<float>(originalHeight));
   }

   wicFactory->CreateBitmapScaler(&scaler);
   scaler->Initialize(
         bitmapSource, destinationWidth, destinationHeight, WICBitmapInterpolationModeCubic);
   converter->Initialize(
         scaler, GUID_WICPixelFormat32bppPBGRA, WICBitmapDitherTypeNone,
         nullptr, 0.f, WICBitmapPaletteTypeMedianCut);
}
else // Don't scale the image.
{
   converter->Initialize(
      bitmapSource, GUID_WICPixelFormat32bppPBGRA, WICBitmapDitherTypeNone,
      nullptr, 0.f, WICBitmapPaletteTypeMedianCut);
```

```
}
```

If the caller of the **LoadBitmapFromFile** method provides either a new width or a new height then a scalar object is created. If just one dimension is specified then the new image is scaled maintaining the aspect ratio of the image. If you want to change the aspect ratio then you can provide both the new height and the new width. Once the code has determined the new height and width, it creates the scalar object through the WIC factory object by calling the **IWICImagingFactory::CreateBitmapScalar** [http://msdn.microsoft.com/en-us/library/ee690296(v=VS.85).aspx] method. This scalar object is then initialized by calling the **IWICBitmapScalar::Initialize** [http://msdn.microsoft.com/en-us/library/ee690169(v=VS.85).aspx] method providing the bitmap source (in this case the frame decoder) and the scaling parameters. The final parameter of this method indicates how the scaling is performed and in this case bicubic interpolation is used. Finally, the pixel format converter is initialized with the scalar, rather than the frame decoder, to add scaling when the image is accessed.

Drawing Images with the WIC

The WIC allows you to perform basic manipulation of an image: scaling, cropping, rotation, and mirroring. If you want to do more complex manipulations then you have to change the individual pixels yourself. Hilo does this by creating a Direct2D render target based on the image you load with the WIC. When you draw on an image, crop it, rotate, or mirror it, the action will be carried out on the Direct2D render target rather than the WIC bitmap. In Hilo, when you save the changes a copy is made of the original and then the changed image is saved to disk.

When started, Annotator locates the image folder and the specified photo in the folder. Information about the photo being edited is stored in an instance of the class **SimpleImage**. This class holds an instance of the **IShellItem** reference to the shell item, so that it knows the location of the original file. It has an **IWICBitmap** reference on the object that the WIC loaded, which is the original image, and an **ID2D1Bitmap** reference to the Direct2D converted from the WIC bitmap which is where changes are made.

In addition there are two vectors, **m_imageOperations** and **m_redoStack**. Each item in these vectors is an interface pointer of the type **IImageOperation** and there are three classes that implement this interface: **DrawGeometryOperation**, **ImageClippingOperation**, and **ImageTransformationOperation**. When you perform a mutable operation on the image, an instance of one of these classes is created and initialized with details of the operation. and stored in the **m_imageOperations** vector so that Annotator has a list of operations that have been performed.

When you draw on an image, this information is stored in the **m_imageOperations** vector as a **DrawGeometryOperation** object. This object has information about the color and the pen width, and a vector of the points visited during the operation. To draw the image, the location information is used to create an instance of the Direct2D geometry object in the **DrawGeometryOperation::UpdateGeometry** method by drawing a Bézier segment between each point and adding these Bézier curves to an **ID2D1PathGeometry** [http://msdn.microsoft.com/en-us/library/dd371512(VS.85).aspx] object.

What you see on screen is the result of applying the operations in the **m_imageOperations** vector, in order, on the **IWICBitmap** object. When you click the undo button the last item is removed from the **m_imageOperations** vector and placed at the top of the **m_redoStack** vector. The **m_redoStack** vector is cleared whenever you perform another operation, so it means that only during the time between undoing an operation and performing another operation can you redo an operation.

When the image is drawn on screen, for example, in response to the user resizing the Annotator window, the method **SimpleImage::DrawImage** is called. This method first draws the **ID2D1Bitmap** object, which is converted from the **IWICBitmap** image loaded by WIC, to the render target of the image editor. Then the **DrawImage** method iterates through each of the drawing operations and applies each one to the render target, and pen drawing operations are applied by calling the **ID2D1RenderTarget::DrawGeometry** [http://msdn.microsoft.com/en-us/library/dd371890(v=VS.85).aspx] method using the **ID2D1PathGeometry** object.

Saving Images with the WIC

In Hilo there are two steps to saving images. The first step is to create a WIC bitmap with the changes that you have performed in the image editor. The second step is to save the WIC bitmap to disk.

Creating a WIC Bitmap

When you save an image, the command handler calls the **SimpleImage::Save** method. The first action of this method is to create a WIC bitmap that will eventually be saved to disk. Listing 4 shows the code to do this. First the code determines the size of the new image from the dimensions of the cropped image and from information about any rotations that have been applied. Second, the code calls the **IWICImagingFactory::CreateBitmap** [http://msdn.microsoft.com/en-us/library/ee690282(v=VS.85).aspx] method using this size. It is important to note that the WIC bitmap is created with the same pixel format (**GUID_WICPixelFormat32bppBGR**) as the Direct2D bitmap that is used.

Listing 4 Creating a new WIC bitmap

C++

```
ComPtr<IWICImagingFactory> wicFactory;
ComPtr<IWICBitmap> wicBitmap;

Direct2DUtility::GetWICFactory(&wicFactory);

// Adjust height and width based on current orientation and clipping rectangle
float width = m_isHorizontal ?
   Direct2DUtility::GetRectWidth(m_clipRect) : Direct2DUtility::GetRectHeight(m_clipRect);
float height = m_isHorizontal ?
   Direct2DUtility::GetRectHeight(m_clipRect) : Direct2DUtility::GetRectWidth(m_clipRect);

wicFactory->CreateBitmap(
   static_cast<unsigned int>(width), static_cast<unsigned int>(height),
   GUID_WICPixelFormat32bppBGR,
   WICBitmapCacheOnLoad, &wicBitmap);
```

The bitmap created in Listing 4 is empty, so the **SimpleImage::Save** method must now construct the bitmap from the original and the drawing operations. The first step is to create a render target based on the WIC bitmap and then load the original image into that render target. This is shown in Listing 5. First, this code calls the **ID2D1Factory::CreateWicBitmapRenderTarget** [http://msdn.microsoft.com/en-us/library/dd371309(v=VS.85).aspx] method, passing the WIC bitmap that was created in Listing 4. The render target returned is like any other render target: you can draw in it using Direct2D resources. The difference is that the pixels that are changed are in the WIC bitmap, rather than the screen. The final part of Listing 5 loads the original image file into the render target so that the drawing operations will be applied to this image.

Listing 5 Creating a render target based on a WIC bitmap

C++

```
ComPtr<ID2D1Factory> d2dFactory;
ComPtr<ID2D1RenderTarget> wicRenderTarget;
Direct2DUtility::GetD2DFactory(&d2dFactory);

d2dFactory->CreateWicBitmapRenderTarget(
   wicBitmap, D2D1::RenderTargetProperties(), &wicRenderTarget);
Direct2DUtility::LoadBitmapFromFile(
       wicRenderTarget, m_imageInfo.fileName.c_str(), 0, 0, &m_bitmap);
```

The next part of the **SimpleImage::Save** method performs the drawing operations that were stored in the **m_imageOperations** vector, Listing 6. The first part creates a transformation so that any rotations performed on the image will be applied to the center of the cropped image. The significant code in Listing 6 is at the end. The **Save** method changes the instance variable **m_currentRenderTarget** to be the render target based on the WIC bitmap and calls the **SimpleImage::DrawImage** method. This means that the drawing operations are applied to the WIC render target and also on the WIC bitmap.

Listing 6 Applying the drawing operations on the WIC bitmap

C++

```
// Get the original bitmap rectangle in terms of the current crop
D2D1_RECT_F originalBitmapRect = D2D1::RectF(
   0, 0, Direct2DUtility::GetRectWidth(m_clipRect),
   Direct2DUtility::GetRectHeight(m_clipRect));

// When rotating images make sure that the point around which rotation occurs lines
// up with the center of the rotated render target
if (false == m_isHorizontal)
{
   float offsetX;
   float offsetY;

   if (width > height)
   {
      offsetX = (width - height) / 2;
      offsetY = -offsetX;
   }
   else
   {
      offsetY = (height - width) / 2;
      offsetX = - offsetY;
   }

   D2D1_MATRIX_3X2_F translation = D2D1::Matrix3x2F::Translation(offsetX, offsetY);
   wicRenderTarget->SetTransform(translation);
}

// Update current render target to point to WIC render target
m_currentRenderTarget = wicRenderTarget;

// Draw updated image to WIC render target
wicRenderTarget->BeginDraw();
DrawImage(originalBitmapRect, m_clipRect, true);
wicRenderTarget->EndDraw();
```

The remainder of the **Save** method makes a backup copy of the original file in a subfolder called AnnotatorBackup and then calls the **Direct2DUtility::SaveBitmapToFile** method to save the WIC bitmap.

Using WIC Encoders

Writing image data is a little more complicated than reading it. You need to have an encoder so that the data is written to the file in the correct format, but since you will be potentially be writing large amounts of data you will also need to initialize the encoder to use block transfers of data, both of the pixels and any metadata that needs to be copied.

The **IWICBitmapEncoder** [http://msdn.microsoft.com/en-us/library/ee690110(v=VS.85).aspx] interface is the starting point for encoding, and the WIC API provides implementations for all the common file formats. To create an encoder you call the **IWICImagingFactory::CreateEncoder** [http://msdn.microsoft.com/en-us/library/ee690311(v=VS.85).aspx] passing a GUID identifying the required encoder. You must then initialize the encoder with a call to the **IWICBitmapEncoder::Initialize** [http://msdn.microsoft.com/en-us/library/ee690123(v=VS.85).aspx] method and provide a stream to the file where the data will be saved. After this you can call the various **Set** methods on the encoder to write data such as the palette or a thumbnail for the image. When you have finished writing to the encoder and want to close the stream and the file it is based upon, you must call the **IWICBitmapEncoder::Commit** [http://msdn.microsoft.com/en-us/library/ee690114(v=VS.85).aspx] method.

The actual pixel data is provided as one or more frame objects, so you must call **IWICBitmapEncoder::CreateNewFrame** [http://msdn.microsoft.com/en-us/library/ee690116(v=VS.85).aspx] for each frame to obtain an **IWICBitmapFrameEncode** [http://msdn.microsoft.com/en-us/library/ee690141(v=VS.85).aspx] interface for the specific frame. This interface has a collection of **Set** methods to set data for the frame such as the size of the image, the resolution, or the pixel format. You write the

actual pixel data through calls to either the **IWICBitmapFrameEncode::WritePixels** [http://msdn.microsoft.com/en-us/library/ee690158(v=VS.85).aspx] method (to write an individual scan line with a buffer of bytes) or the **IWICBitmapFrameEncode::WriteSource** [http://msdn.microsoft.com/en-us/library/ee690159(v=VS.85).aspx] method (to write the entire frame from an **IWICBitmapSource** object). When you have finished writing a frame you must call **IWICBitmapFrameEncode::Commit** [http://msdn.microsoft.com/en-us/library/ee690142(v=VS.85).aspx] method to indicate that the frame can be written to the stream.

As mentioned earlier, a frame may have metadata and you can obtain a **IWICMetadataQueryWriter** [http://msdn.microsoft.com/en-us/library/ee719717(v=VS.85).aspx] object by calling the **IWICBitmapFrameEncode::GetMetadataQueryWriter** [http://msdn.microsoft.com/en-us/library/ee690144(v=VS.85).aspx] method. However, if you already have a bitmap source object with metadata you can copy the metadata as a block by using metadata block readers and writers. The frame decoder supports block metadata reading through the **IWICMetadataBlockReader** [http://msdn.microsoft.com/en-us/library/ee690327(v=VS.85).aspx] interface. Normally you would not use this interface directly; instead you use it to initialise a metadata block writer. A frame encoder implements the **IWICMetadataBlockWriter** [http://msdn.microsoft.com/en-us/library/ee690335(v=VS.85).aspx] interface and you can initialize the block writer by calling the **IWICMetadataBlockWriter::InitializeFromBlockReader** [http://msdn.microsoft.com/en-us/library/ee690338(v=VS.85).aspx] method passing the block reader object. This method will then read all the metadata and write them to the image file stream when the frame is committed.

Saving a WIC Bitmap in Annotator

In Hilo the Common project provides the **Direct2DUtility::SaveBitmapToFile** method to obtain an encoder and save an image to disk. The first part of this method needs to get information about the file that it is creating, so first it calls **IWICImagingFactory::CreateDecoderFromFilename** to get a decoder for the original file and through this object the **SaveBitmapToFile** method can determine how many frames there are in the original image file. Annotator only allows you to edit the first frame, and an image file may contain more than one frame, so when saving the altered file the first frame will be the image altered in Annotator and any other frames are simply copied from the original file.

The next part of the method is to create an encoder, and this code is shown in Listing 7. To create an encoder using the factory object you have to supply the encoder type and the **SaveBitmapToFile** method determines the type from the file extension.

Listing 7 Creating an encoder

C++

```
GUID containerFormat = GUID_ContainerFormatBmp;
ComPtr<IWICImagingFactory> factory;
ComPtr<IWICBitmapEncoder> encoder;
Direct2DUtility::GetWICFactory(&factory);

// File extension to determine which container format to use for the output file
std::wstring fileExtension(outputFilePath.substr(outputFilePath.find_last_of('.')));
// Convert all characters to lower case
std::transform(fileExtension.begin(), fileExtension.end(), fileExtension.begin (), tolower);

// Default value is bitmap encoding
if (fileExtension.compare(L".jpg") == 0 ||
    fileExtension.compare(L".jpeg") == 0 ||
    fileExtension.compare(L".jpe") == 0 ||
    fileExtension.compare(L".jfif") == 0)
{
   containerFormat = GUID_ContainerFormatJpeg;
}
else if (fileExtension.compare(L".tif") == 0 ||
          fileExtension.compare(L".tiff") == 0)
{
   containerFormat = GUID_ContainerFormatTiff;
}
```

```
// Other code omitted

factory->CreateEncoder(containerFormat, nullptr, &encoder);
```

After creating the encoder you must initialize it with a stream that indicates where the data is to be written. Listing 8 shows the code to do this. The factory object is used to create a stream and the stream is initialized with the path to the output file. The stream is then used to initialize the encoder object.

Listing 8 Initializing the encoder

C++

```
ComPtr<IWICStream> stream;
// Create a stream for the encoder
factory->CreateStream(&stream);
// Initialize the stream using the output file path
stream->InitializeFromFilename(outputFilePath.c_str(), GENERIC_WRITE);
// Create encoder to write to image file
encoder->Initialize(stream, WICBitmapEncoderNoCache);
```

At this point you have three objects: the decoder with the original data, the WIC bitmap with the changed bitmap, and the encoder that is initialized with a stream to the location to store the data. The final part of the **SaveBitmapToFile** method is to write the WIC bitmap as the first frame to the encoder and if the original file has more than one frame then all the other frames are read from the decoder and written to the encoder. Each time the **SaveBitmapToFile** method writes a frame it must write the metadata for that frame and the simplest way to do this is through metadata block readers and writers.

Listing 9 shows the first part of the loop to write a frame. The first part of this code obtains the frame decoder for the specified frame and creates a new frame in the encoder, it then sets the size of the new frame and if this is the first frame the size is taken from the WIC bitmap. Otherwise it is taken from the frame read from the decoder from the original file. Next, the code reads the pixel format of the frame from the original file and uses this to set the frame encoder.

Listing 9 Creating a frame using an encoder

C++

```
ComPtr<IWICMetadataBlockWriter> blockWriter;
ComPtr<IWICMetadataBlockReader> blockReader;

for (unsigned int i = 0; i < frameCount && SUCCEEDED(hr); i++)
{
   //Frame variables
   ComPtr<IWICBitmapFrameDecode> frameDecode;
   ComPtr<IWICBitmapFrameEncode> frameEncode;

   //Get and create image frame
   decoder->GetFrame(i, &frameDecode);
   encoder->CreateNewFrame(&frameEncode, nullptr);

   //Initialize the encoder
   frameEncode->Initialize(NULL);

   //Get and set size
   if (i == 0)
   {
      updatedBitmap->GetSize(&width, &height);
   }
   else
   {
      frameDecode->GetSize(&width, &height);
   }
```

```
    frameEncode->SetSize(width, height);

    //Set pixel format
    frameDecode->GetPixelFormat(&pixelFormat);
    frameEncode->SetPixelFormat(&pixelFormat);
```

The final part of the loop reads the data for the frame and writes the data through the encoder, as shown in Listing 10. First the code obtains the metadata block reader from the decoder and uses this to initialize the metadata block writer from the encoder so that the frame's metadata is written in one action. Then the frame pixel data is written in one action with a call to **IWICBitmapFrameEncode::WriteSource** and passing either the WIC bitmap or the frame decoder object. The final part of the loop is to commit the frame, and then when all frames have been written, there is a call to **IWICBitmapEncoder::Commit** to write the entire image to the file.

Listing 10 Writing a frame with an encoder

C#

```
    //Check that the destination format and source formats are the same
    bool formatsEqual = false;
    GUID srcFormat;
    GUID destFormat;

    decoder->GetContainerFormat(&srcFormat);
    encoder->GetContainerFormat(&destFormat);
    formatsEqual = (srcFormat == destFormat) ? true : false;

    if (formatsEqual)
    {
       //Copy metadata using metadata block reader/writer
       frameDecode->QueryInterface(&blockReader);
       frameEncode->QueryInterface(&blockWriter);

       if (nullptr != blockReader && nullptr != blockWriter)
       {
          blockWriter->InitializeFromBlockReader(blockReader);
       }
    }

    if (i == 0)
    {
       // Copy updated bitmap to output
       frameEncode->WriteSource(updatedBitmap, nullptr);
    }
    else
    {
       // Copy existing image to output
       frameEncode->WriteSource(static_cast<IWICBitmapSource*> (frameDecode), nullptr);
    }

    //Commit the frame
    frameEncode->Commit();
 }
 encoder->Commit();
```

Conclusion

The Windows 7 Imaging Component allows you to load images with standard image formats from files, resources, and streams. It will decode the image data into a format that you can use with GDI or Direct2D. Hilo uses the WIC in both the Browser application and the Annotator application to display images, and the Annotator application uses the WIC to save edited photos to disk. In the next chapter we will introduce the Hilo Share application, which allows you to share your images online.

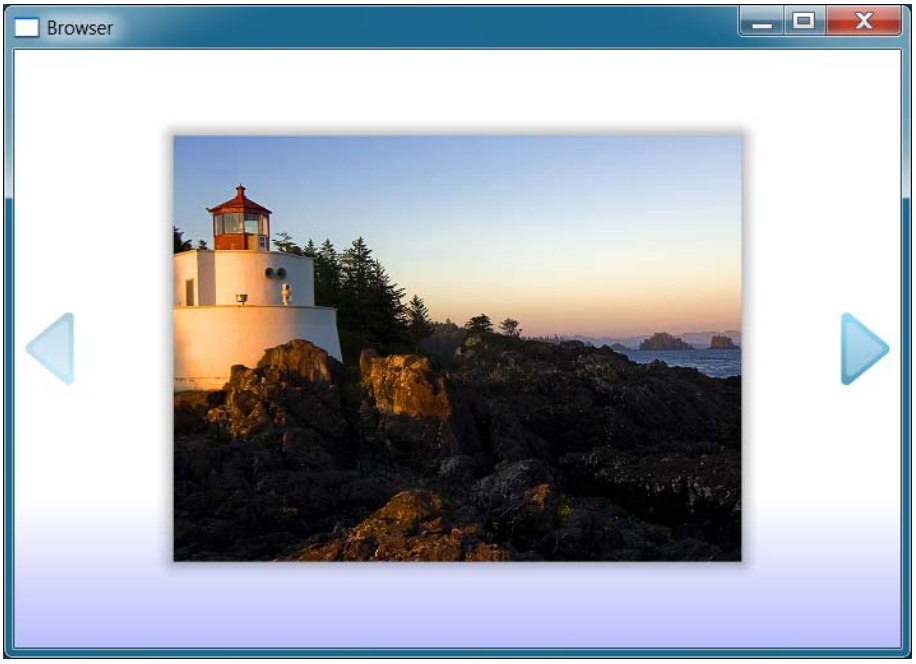
Chapter 12: Sharing Photos with Hilo

In this chapter, we'll describe how the Hilo applications have been extended to allow you to share photos via an online photo sharing site. To do this, Hilo uses the Windows 7 Web Services application programming Interface (WSSAPI). The Hilo Browser application has also been updated to provide additional user interface (UI) and touch screen features, and the Hilo Annotator application has been extended to support Windows 7 Taskbar Jump Lists. You can download the final version of the Hilo source code from here [http://go.microsoft.com/?linkid=9730262] . This chapter provides an overview of these new features.

Updating the Hilo Browser User Interface

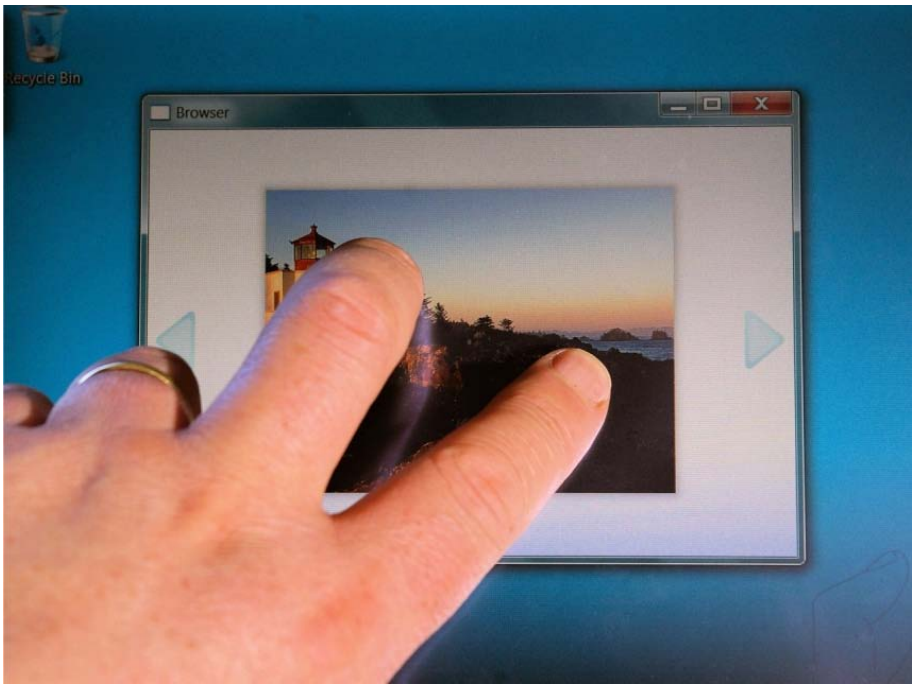
Hilo's photo sharing functionality is accessed through the Hilo Browser application. Previous versions of the Hilo Browser allowed the user to launch the Annotator by double-tapping (or double-clicking) a photo in the media pane. This worked adequately because there was just one action that could be performed on a photo. Now that there is an additional action—share—another approach must be used. In the final version of the Hilo Browser, double-tapping a photo shows the photo in the slideshow mode (Figure 1) while double-tapping the screen again returns to browsing mode.

Figure 1 Using the slide show mode



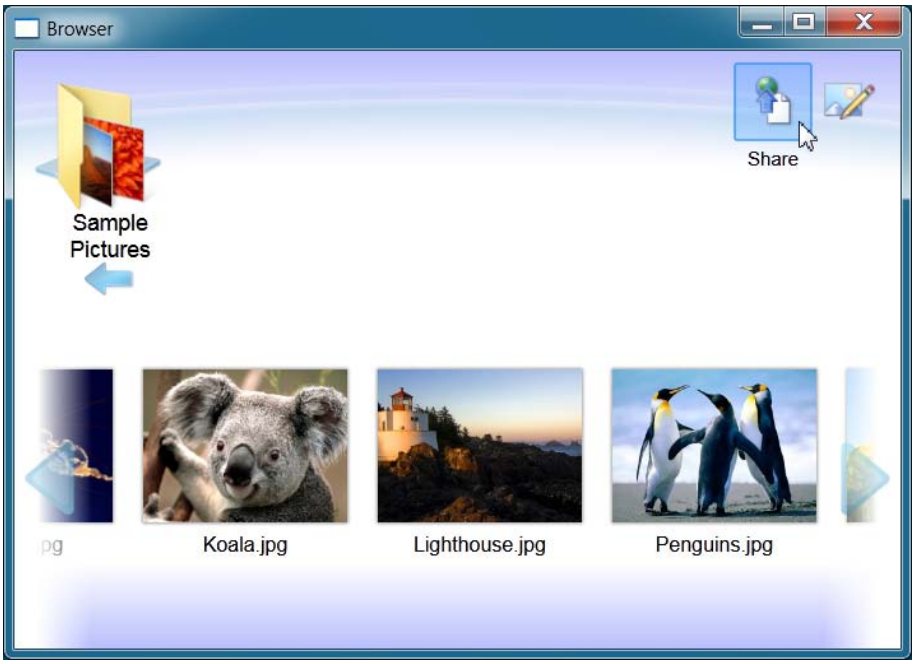
In slideshow mode, the carousel disappears and the selected photo is maximized to fill the window. Only one photo is shown in slideshow mode. Arrows are shown on either side of the photo to allow you to pan to the photo to the left or right. On a touch screen you can also touch the photo and, with your finger still on the image, flick it left or right to scroll in that direction. This is known as a touch screen gesture. The Hilo Browser supports two types of gestures: pan (the flick gesture to change photo) and zoom.

Figure 2 Resizing a photo with the zoom gesture



The zoom gesture requires two fingers (Figure 2) touching the screen. When you move your fingers apart to zoom in, the image size increases, and when you move your fingers together to zoom out, the image size decreases. Gestures like this can make an application feel more natural to use.

Figure 3 The Share and Annotate Buttons in the Hilo Browser



The Hilo Browser, in browsing mode, now also provides two buttons in the top right corner, as shown in Figure 3. These buttons allow you to launch the Annotator application, or to share the selected photos via an online photo sharing site. These buttons display images to illustrate their purpose: the **Share** button has an arrow directing the movement of the photo document to the globe representing the Internet; the **Annotator** button has a photo and a pencil. Normally these buttons do not have captions, but when you hover the mouse cursor over a button, the button edge is highlighted and the caption appears beneath the button.

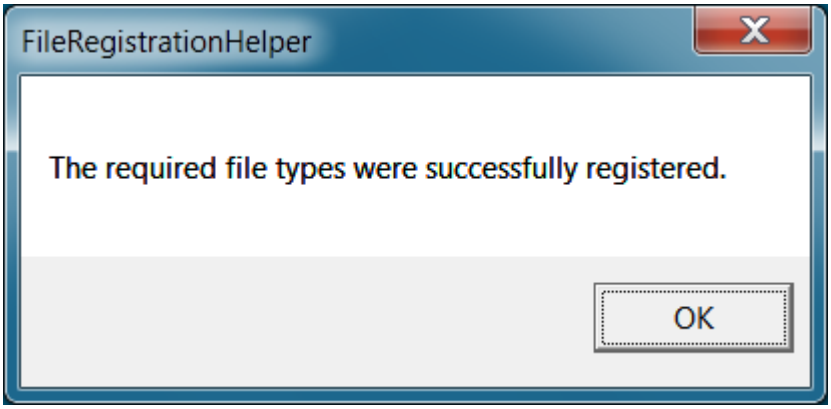
Using Taskbar Jump Lists

The final version of the Hilo Annotator application adds support for Windows 7 Taskbar Jump Lists, which allows you to easily access previously edited photos and to initiate important tasks directly from the taskbar. In order to

uses these features, however, your application will have to be registered with the Windows shell. Hilo provides a utility application to help with this process.

When the Annotator first runs (for example, in response to you clicking on the **Annotator** button in the Browser) it checks to see if the application is registered and if not, it runs a utility called **RegistrationHelper**. This is a once-only action. **RegistrationHelper** is a console application and it needs Administrator privileges to run. If your account does not have these privileges then you will get an error message. Even if you run **RegistrationHelper** as Administrator you will get a User Access Control message asking you if the utility should be given permission to run (click **Yes**). After running the registration helper utility, it will inform you if the action was successful (Figure 4). From this point onwards, the Hilo Annotator application can display the **Recent** files Jump Lists on its Taskbar.

Figure 4 Successful registration



The utility can also be used to unregister Annotator, for example if you want to remove Hilo from your computer. The command line help is shown in Listing 1. You have to provide at least six command line parameters. The first parameter determines whether you wish to register or unregister the application (use the string **TRUE** or **FALSE**). The next parameter is a ProgID (for Annotator use **Microsoft.Hilo.AnnotatorProgID**) and the utility will either add or remove this key from the **HKEY_CLASSES_ROOT** registry hive. The next three parameters are only needed if you are registering Annotator. The third parameter is the full path to the Annotator application, the fourth is the name that will be used to identify that a file type is edited by Annotator and the fifth parameter is an identifier called an application user model ID, which is essentially used to group together icons for an application as a single button on the taskbar (for Annotator use **Microsoft.Hilo.Annotator**). The remaining command line parameters are the extensions of the file types that Annotator will edit.

Listing 1 Command line options for RegistrationHelper.exe

```
Usage: RegistrationHelper <Register:TRUE|FALSE> <ProgID> <FullPath><FriendlyDocumentName>
<AppUserModelId> <ext1,ext2,ext3,...>
```

If you wish to unregister the application then you must provide the ProgID and the file extensions, but you can provide empty strings for the other parameters, this is shown in Listing 2, where empty strings (“”) are given for the path, the friendly name and the **AppUserModelID** parameter.

Listing 2 Unregistering Annotator

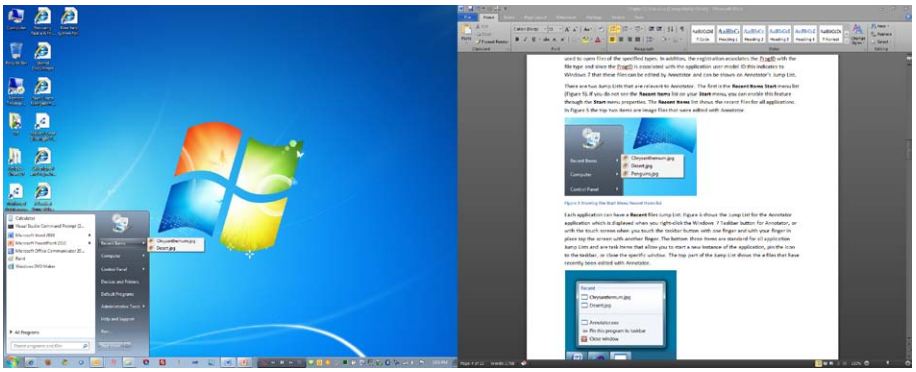
```
RegistrationHelper FALSE Microsoft.Hilo.AnnotatorProgID "" "" "" .bmp .dib .jpg .jpeg .jpe
.jfif .gif .tif .tiff .png
```

The registration utility adds values to the registry to indicate that the Annotator application may be used to open files of the specified types. In addition, the registration associates the ProgID with the file type and since the ProgID is associated with the application user model ID this indicates to Windows 7 that these files can be edited by Annotator and can be shown on Annotator’s Jump List.

There are two **Recent Item** lists that are relevant to Annotator. The first is the **Start** menu **Recent Items** list

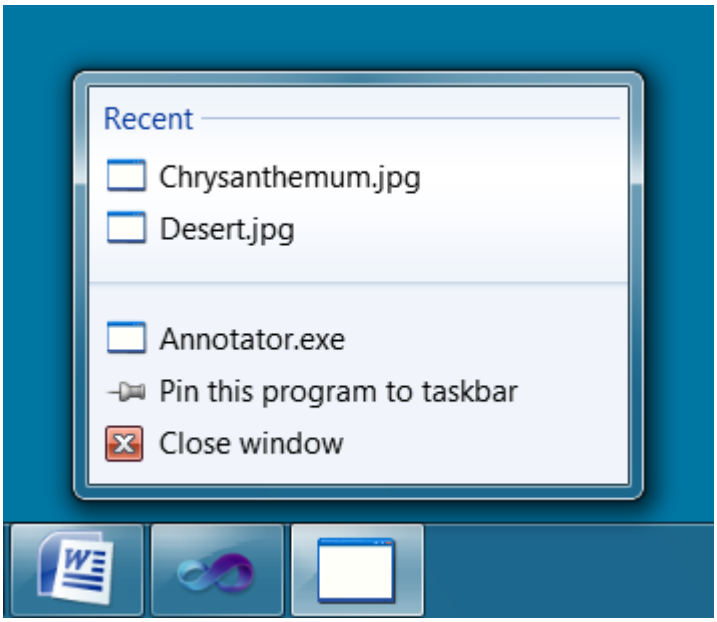
(Figure 5). If you do not see the **Recent Items** list on your **Start** menu, you can enable this feature through the **Start** menu properties. The **Recent Items** list shows the recent files for all applications. In Figure 5 the two recent items are image files that were edited with Annotator.

Figure 5 Showing the Start Menu Recent Items list



Each application can have a **Recent** files Jump List. Figure 6 shows the Jump List for the Annotator application which is displayed when you right-click the Windows 7 Taskbar button for Annotator, or with the touch screen when you touch the taskbar button with one finger and with your finger in place tap the screen with another finger. The bottom three items are standard for all application Jump Lists and are task items that allow you to start a new instance of the application, pin the icon to the taskbar, or close the specific window. The top part of the Jump List shows the files that have recently been edited with Annotator.

Figure 6 Showing the Jump List for Annotator



If you compare Figure 5 with Figure 6 you can see that the files added to the Annotator **Recent** file list are added to the **Start** menu **Recent Items** list. When Annotator adds a file to the **Recent** file list it associates the file with the application user model ID (often abbreviated as AppID, but this is *not* the same as the GUID used by processes to specify DCOM parameters). The AppID allows the icons of multiple instances of the same application to be grouped together on the taskbar, however if you choose you can give different instances different AppIDs to provide separate icons on the taskbar for each different AppID.

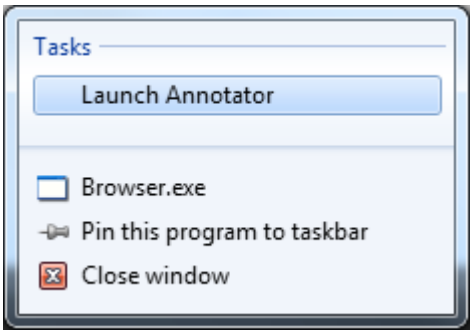
When you click on an item in the **Recent** file list, Windows 7 locates the ProgID for the AppID for the button, and uses the **open** verb defined in the ProgID to start a new instance of the application with the item. If instead, you right-click the item you'll get a context menu which includes the option to view the property page for the item, Figure 7. This shows that the file type (the friendly type name) is **Microsoft Hilo Annotator**, rather than a generic description of the image type. The important point is that this file type is displayed only when the properties dialog is accessed through the Annotator Jump List. If the file's properties are shown through Windows

Explorer then the default file type will be displayed.

Figure 7 Property page for an item on the Annotator Jump List

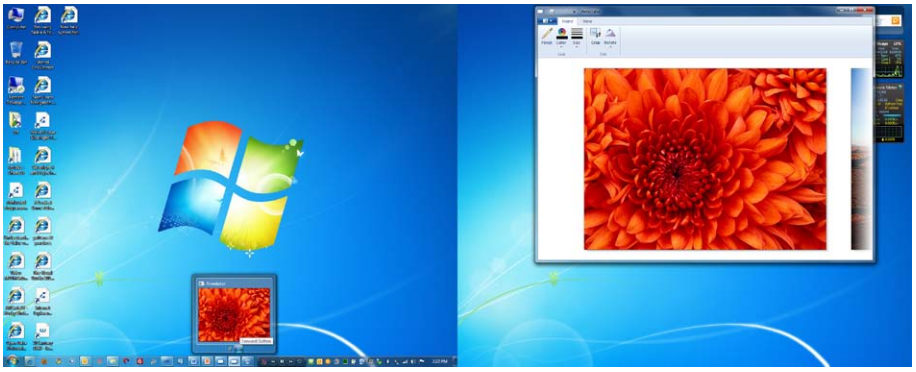
The **Recent** file list is one of the standard categories that you can use on an application's Jump List. Figure 6 shows the standard tasks that are appended to all Jump Lists. You can add custom tasks and Browser adds a custom task to launch Annotator, Figure 8. In this case the task is a shell link object that starts the Annotator process.

Figure 8 Browser custom task list



In Windows 7 the default action of hovering the mouse over (or holding your finger on) a taskbar button gives a thumbnail image of the application's window but developers can customize this view and even add controls to it. Hilo Annotator implements a custom preview window, as illustrated in Figure 9. The Annotator does two things: first it provides a preview of the selected photo, and second it provides two buttons which you can click to change the thumbnail (and the image in Annotator) with the image to the left or right.

Figure 9 Annotator task bar preview



Sharing Photos on Flickr

The Hilo Browser allows you to share photos through Flickr—a popular internet photo sharing site that implements a Web Service API to allow photos to be published programmatically. The Hilo Browser uses the Windows 7 Web Services Application Programming Interface (WWSAPI) to access the Flickr API to authenticate a user and to upload a photo to the user's photostream.

In order for any application to be able to upload photos to Flickr, it must first be registered with Flickr. Flickr then issues a two part **Flickr API Key**. One part of this key is used to identify the application, while the other part is used to sign and authenticate the data that it passes through the Flickr API. The Hilo Browser application passes this key to Flickr whenever it performs an action against the Flickr API.

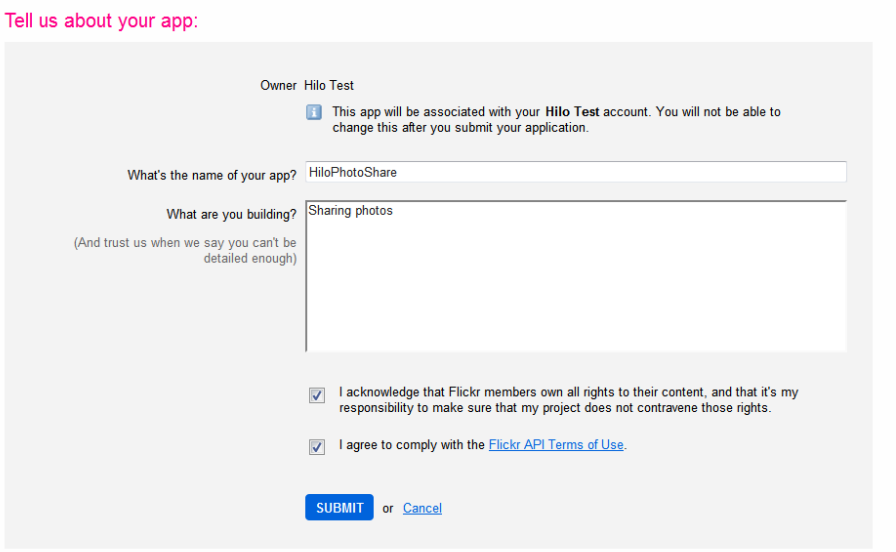
Once the Hilo application has been registered with Flickr, it can be used by any user to upload photos to their account. To enable this, however, the user has to specifically authorize the Hilo application so that it can access their account. Issuing a Flickr API Key means that Flickr knows about and trusts the Hilo application. But individual users then have to choose whether or not to trust the Hilo application.

Obtaining a Flickr API Key

The Hilo Browser source code does not include a Flickr API key. You will need to obtain a key yourself and add it to the Hilo Browser source code before it can interact with the Flickr API. You can apply for an API key through the Flickr services website.

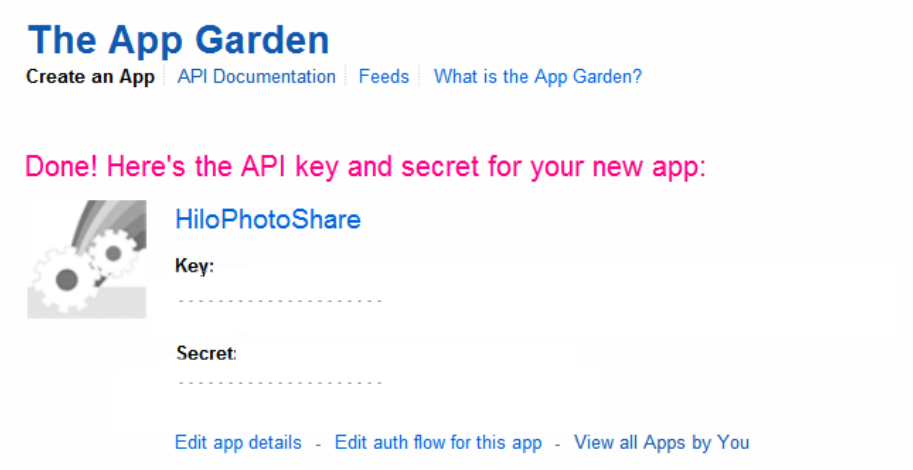
You can use any Flickr account to obtain a Flickr API Key for Hilo. If you don't already have a Flickr account, you first have to create a Yahoo account [https://na.edit.yahoo.com/registration?.pd=flickr_ver] , log in ,and then follow the steps to create a Flickr account [https://login.yahoo.com/config/login?.src=flickr] . Once you have created a Flickr account you can then apply for an API key [http://www.flickr.com/services/apps/create/apply/] . There are two options: a key for commercial applications, or one for non-commercial applications. To test out Hilo, you should apply for a non-commercial key [http://www.flickr.com/services/apps/create/noncommercial/] . This will result in a web page similar to Figure 10, where you are asked to provide information about your application.

Figure 10 Applying for a Flickr API key



Flickr will then provide you with two strings (Figure 11), one is the API key that uniquely identifies the application (Hilo in this case), and the other string is a shared secret string that Hilo will use to sign data sent to Flickr. This enables Flickr to verify the integrity of the parameters passed to it. You should make a note of the API key and secret strings because you will need to add them to the Hilo Browser source code, as described in the next section.

Figure 11 Obtaining the API key for Flickr



Adding the API Key to the Hilo Browser

The Flickr API key identifies the Hilo Browser application to Flickr. It is defined directly in the Hilo Browser's source code so that it can be easily accessed and used whenever a photo is uploaded. Once you have obtained a Flickr API key, you will need to add the two strings to the Hilo Browser source code. To do this open the **FlickrUploader.cpp** file in Visual Studio 2010 and search for the two lines shown in Listing 3. Paste the strings for the Flickr API key and secret into these two variables.

Note:

Storing cryptographic secrets clear text in a code file is NOT recommended for production code. Hackers can simply scan the executable and obtain the key. However, since the purpose of Hilo is to illustrate how to access a Web Service, rather than to illustrate cryptography, the simple (though unsecure) approach is used.

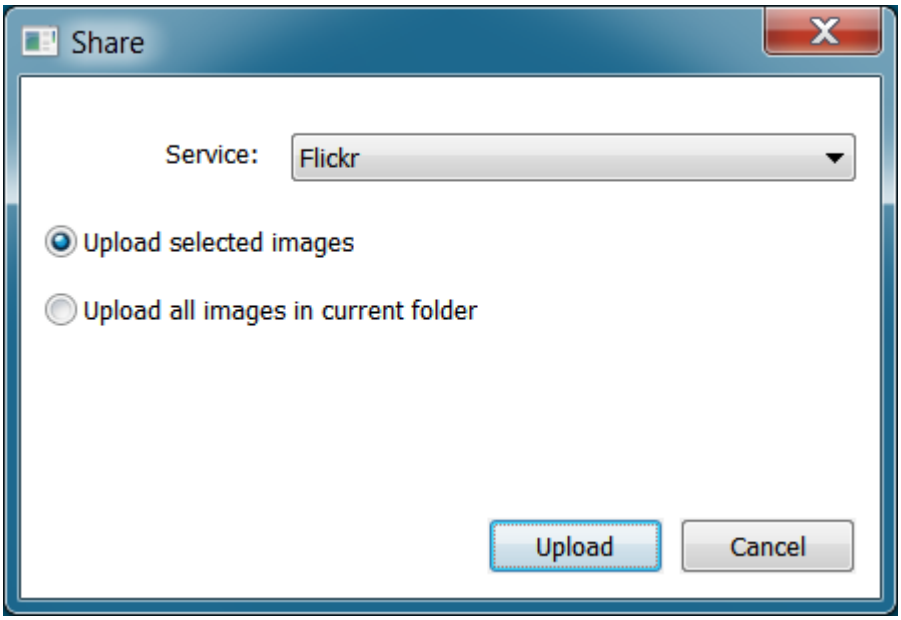
Listing 3 Providing Hilo access to the Flickr API Key

```
static wchar_t* flickr_api_key = L"";
static wchar_t* flickr_secret = L"";
```

Sharing a Photo using the Hilo Browser

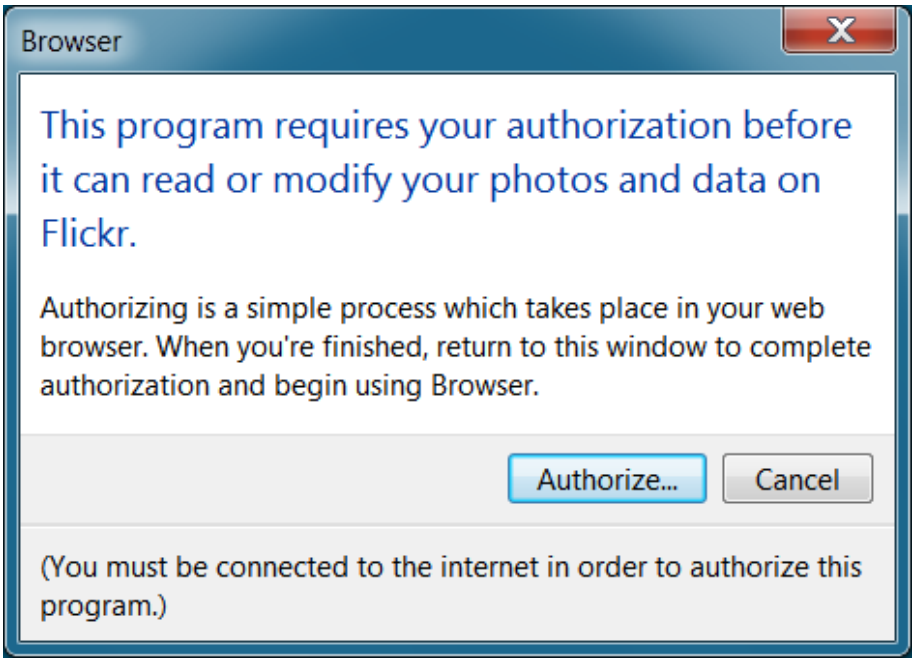
Once you have recompiled the Hilo Browser application you will be able to upload photos to Flickr. To do this, select a photo in the media pane and touch (or click) the **Share** button. This will launch the **Share** dialog shown in Figure 12. This dialog allows you to determine if a single file, or all files in the folder, are uploaded and it allows you to select the photo sharing service that will be used (only Flickr is supported at the moment).

Figure 12 Sharing a photo through Flickr



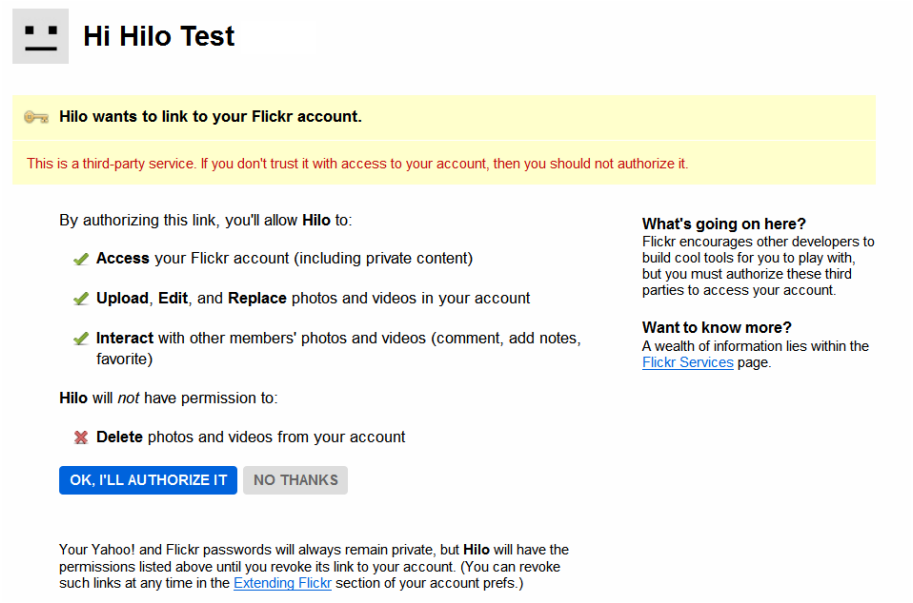
Hilo will upload photos to your account once you acknowledge that Hilo has permission to do this. Once you click the **Upload** button on the **Share** dialog you will see the authorization dialog, Figure 13, requesting that you authorize Hilo.

Figure 13 Authorizing Hilo to upload to your Flickr account



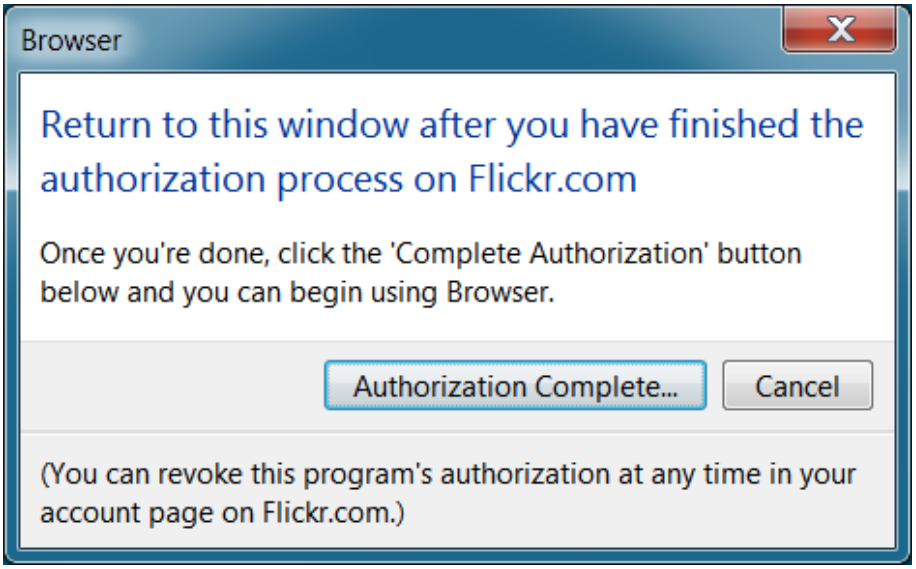
When you click the **Authorize** button Hilo launches your browser and opens the Flickr web page to authorize Hilo. The URL of this web page identifies Hilo by providing the API Key as one of the URL's GET parameters. If you are not logged in to Flickr you will be requested to login in before the authorization web page is shown, Figure 14. This page says that photos will be uploaded to your Flickr account , but before this can be done this account must authorize Hilo.

Figure 14 Authorizing Hilo



There is no authorization callback to Hilo, so if you decide not to authorize Hilo to access your Flickr account Hilo will only find out is when it attempts to upload a photo and the access fails. So regardless of which button you click on the web page shown in Figure 14 you will see the confirmation dialog, Figure 15, which is used simply to block Hilo from uploading until authorization is complete.

Figure 15 Confirming authorization



When you click the **Authorization Complete** button and the Flickr account has authorized Hilo, the upload process starts and progress is shown using the dialog shown in Figure 16. When the upload is complete you will see a message and a link to view the photos that you have uploaded as shown in Figure 17.

Figure 16 Uploading the photo

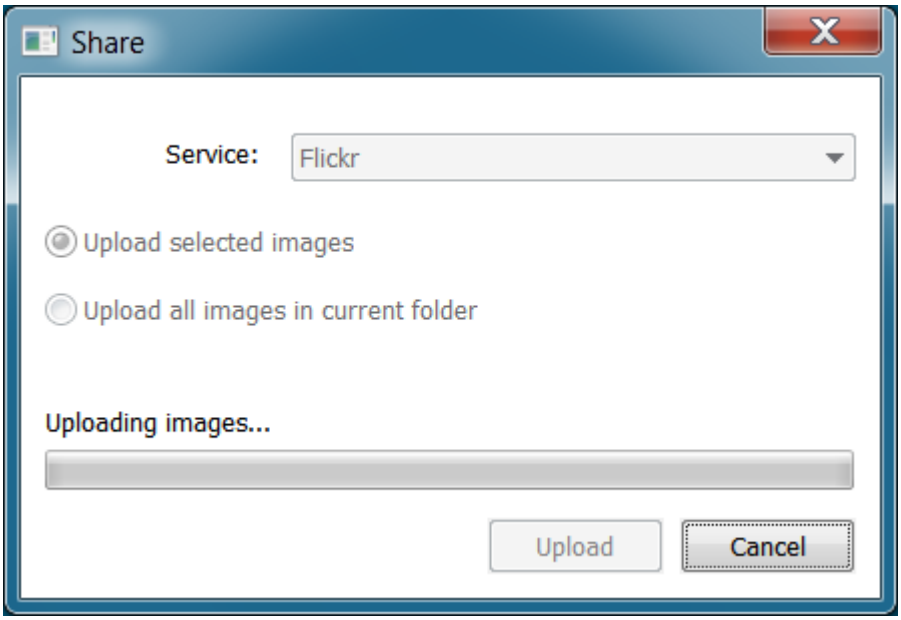
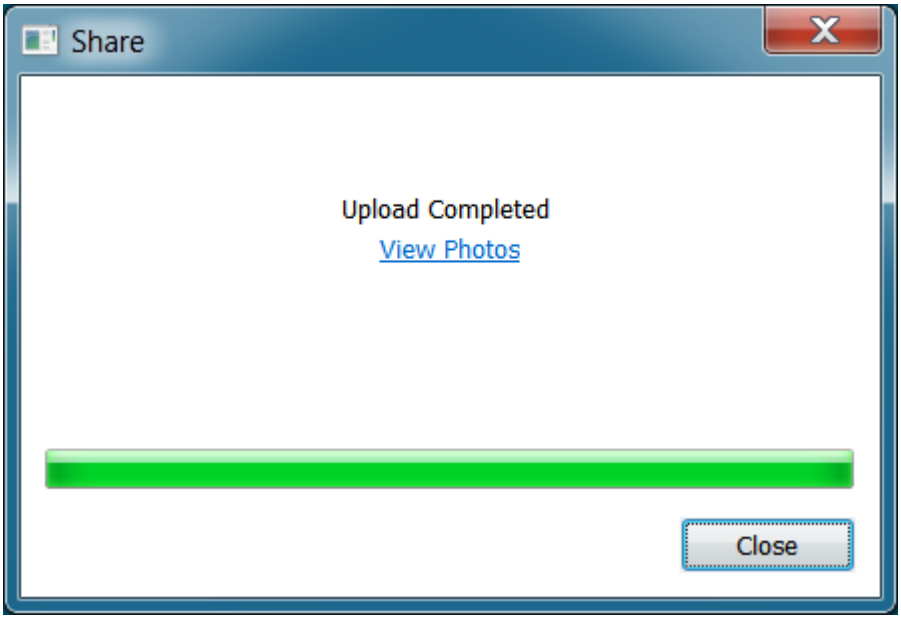
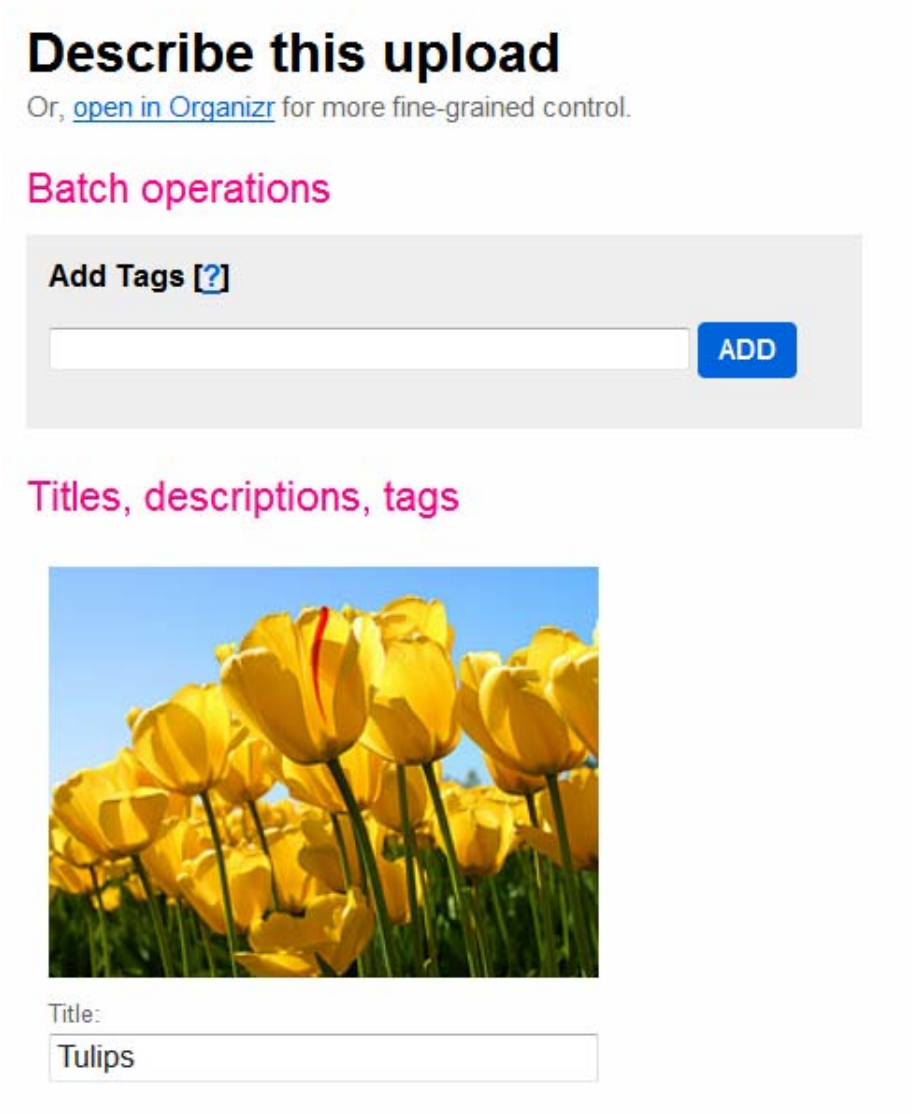


Figure 17 Photo upload finished



The URL link is to the Flickr page which allows you to add a description and tags for the photo, Figure 18.

Figure 18 Uploaded photo web page



Conclusion

The final versions of the Hilo applications add some additional UI features and the ability to upload photos to the Flickr photo sharing website. The Hilo Browser now supports touch screen gestures to pan and zoom photos, while the Hilo Annotator now provides Jump Lists to give quick access to recently edited files and key tasks. The Hilo Browser contains the code to upload photos to Flickr via the Flickr API web service. In the next chapter we'll cover the implementation of the user interface features in more detail.

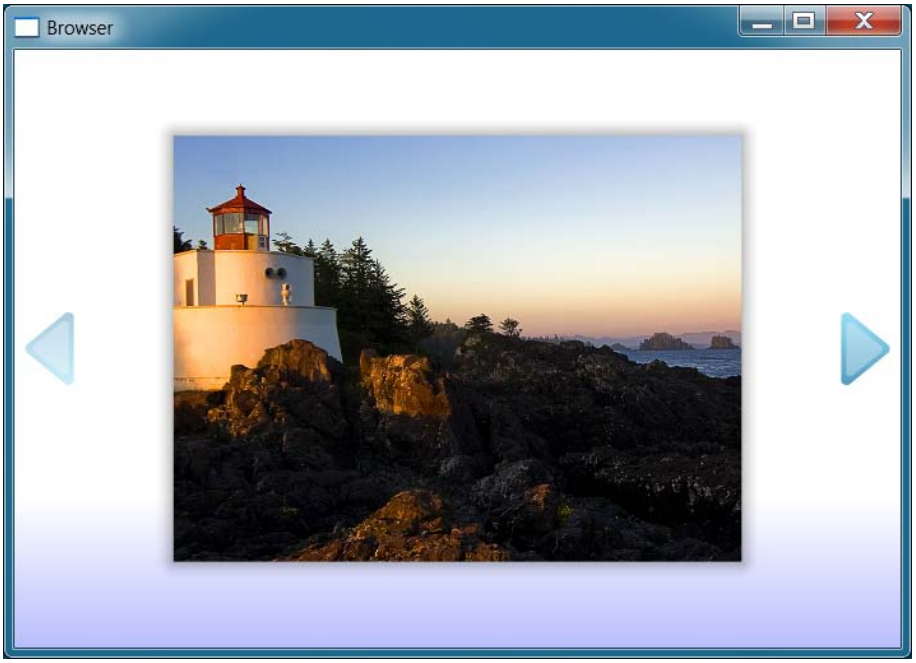
Chapter 13: Enhancing the Hilo Browser User Interface

In the final version of Hilo, the Annotator and Browser applications provide a number of enhanced user interface (UI) features. For example, the Hilo Browser now provides buttons to launch the Annotator application, to share photos via Flickr, and touch screen gestures to pan and zoom images. In this chapter we will see how these features were implemented.

Launching the Share Dialog and Hilo Annotator

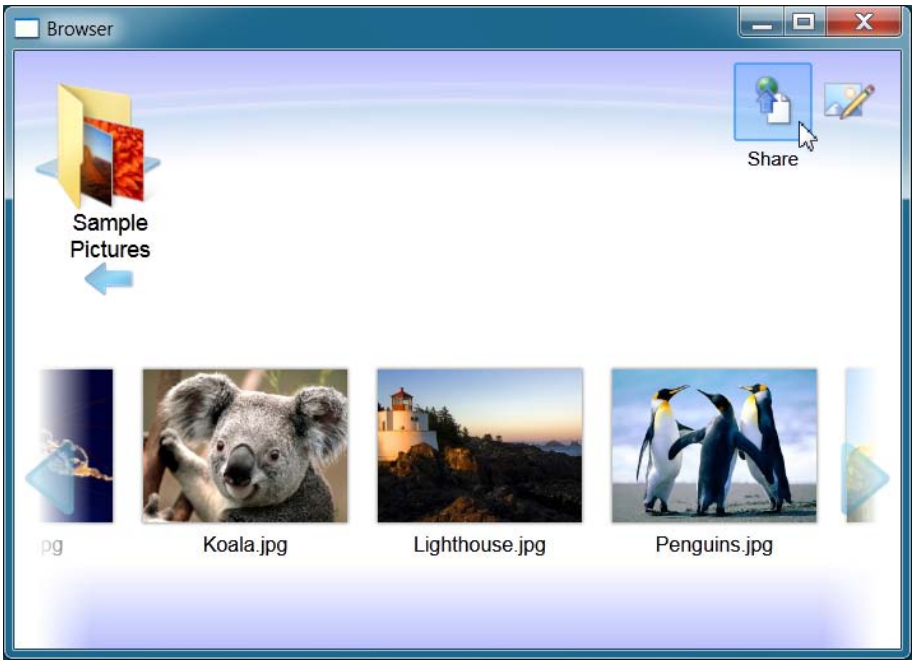
The Hilo Browser application now allows you to share selected photos through an online photo sharing application. It also allows you to edit selected photos by launching the Hilo Annotator application. The Hilo Browser has been extended to make it easier to perform these two actions. In the first version of the Browser double-clicking (or double-tapping) on a photo launches the Annotator in order to edit the photo. The Browser now uses the double-click gesture to launch the slide show mode where the carousel is hidden and the selected photo is shown at a larger scale (Figure 1).

Figure 1 Browser in slideshow mode



The Hilo Browser provides two buttons: one to start the Share dialog and the other to start the Hilo Annotator application (Figure 2). Normally these two buttons appear as icons in the top right hand corner of Browser, but when you hover the mouse cursor over the icon the Browser shows the edge of the button and its caption. These two buttons are not button controls, in fact they are not even child windows, instead the images are Direct2D bitmaps and the click action is performed through hit testing.

Figure 2 Hilo Browser showing the buttons to launch Annotator and Share



The top half of the client area for Browser is implemented by the **CarouselPaneMessageHandler** class. This class has two members called **m_annotatorButtonImage** and **m_sharingButtonImage** which are references to bitmap objects that implement the **ID2D1Bitmap** interface. Both members are initialized in the **CarouselPaneMessageHandler::CreateDeviceResources** method by loading the image from a bitmap resource bound to the Browser process (Figure 3).

Figure 3 Icons for Browser's buttons

When the window is resized the message handler calls the **CarouselPaneMessageHandler::CalculateApplicationButtonRects** method to determine the position of each button image and a rectangle for the selection. When the window is redrawn the **CarouselPaneMessageHandler::DrawClientArea** method is called and this draws the images at these calculated positions using the **ID2D1RenderTarget::DrawBitmap** [http://msdn.microsoft.com/en-us/library/dd371878(v=VS.85).aspx] method. However, an image on its own does not show user feedback nor handle mouse clicks.

The user feedback is provided by testing to see if the mouse has hovered over the button and then drawing the selection rectangle, the following discussion is for the **Annotator** button but it also applies to the **Share** button. The first action, testing to see if the mouse is hovering over the button, is carried out in the **CarouselPaneMessageHandler::CheckForMouseHover** method in response to a mouse move message, and the code in **CheckForMouseHover** is shown in Listing 1. This code calls **Direct2DUtility::HitTest** which simply tests whether the mouse position is within the selection rectangle for the button. The result from the hit test is saved in the Boolean variable **m_isAnnotatorButtonMouseHover** which is used later on in the code.

Listing 1 Hit testing for the Annotator button

C++

```
if (Direct2DUtility::HitTest(m_annotateButtonSelectionRect.rect, mousePosition))
{
   if (!m_isAnnotatorButtonMouseHover)
   {
      needsRedraw = true;
      m_isAnnotatorButtonMouseHover = true;
      CalculateApplicationButtonRects();
   }
}
```

```
else
{
   if (m_isAnnotatorButtonMouseHover)
   {
      needsRedraw = true;
      m_isAnnotatorButtonMouseHover = false;
      CalculateApplicationButtonRects();
   }
}
```

When the carousel window is redrawn the **m_isAnnotatorButtonMouseHover** member variable is checked, and if it indicates that the mouse is hovering over the button the selection rectangle is outlined in a solid color and the rectangle is filled with the same color but with a 25% opacity as shown in Listing 2. There is no code to remove this rectangle when the mouse moves away from the button, in this situation the entire carousel window is redrawn without the selection.

Listing 2 Code to draw the selection box around a button image

C++

```
// Draw selection box for annotate button
if (m_isAnnotatorButtonMouseHover)
{
   m_selectionBrush->SetOpacity(1.0f);
   m_renderTarget->DrawRoundedRectangle(m_annotateButtonSelectionRect, m_selectionBrush);

   m_selectionBrush->SetOpacity(0.25f);
   m_renderTarget->FillRoundedRectangle(m_annotateButtonSelectionRect, m_selectionBrush);

   m_renderTarget->DrawTextLayout(
      D2D1::Point2(m_annotateButtonSelectionRect.rect.left,
m_annotateButtonSelectionRect.rect.bottom),
      m_textLayoutAnnotate,
      m_fontBrush);
}

m_renderTarget->DrawBitmap(m_annotatorButtonImage, m_annotateButtonImageRect);
```

Mouse clicks are handled in a similar way. Listing 3 shows an excerpt from the handler for the **WM_LBUTTONUP** [http://msdn.microsoft.com/en-us/library/ms645608(VS.85).aspx] mouse message. If the Browser history stack is expanded then button clicks are ignored, but if the history stack is not expanded then there is a check to see if the mouse is above one or other of the buttons, and if so either the **MediaPaneMessageHandler::LaunchAnnotator** or **MediaPaneMessageHandler::ShareImages** method is called.

Listing 3 Handling Browser button mouse clicks

C++

```
if (!clickProcessed)
{
   if (m_isHistoryExpanded)
   {
      // other code
   }
   else
   {
      // Check if the user clicked the share or annotate application button
      if (m_isAnnotatorButtonMouseHover || m_isSharingButtonMouseHover)
      {
         ComPtr<IMediaPane> mediaPane;
         hr = m_mediaPane->QueryInterface(&mediaPane);
```

```
            if (SUCCEEDED(hr))
            {
               if (m_isAnnotatorButtonMouseHover)
               {
                  mediaPane->LaunchAnnotator();
               }
               else
               {
                  mediaPane->ShareImages();
               }
            }
         }
      }
   }
```

The **MediaPaneMessageHandler::LaunchAnnotator** was covered in an earlier chapter [http://msdn.microsoft.com/en-us/library/ff951239.aspx] (it simply calls the Windows API function **CreateProcess** to start the Annotator application). The code to share photos is implemented in Browser. The **MediaPaneMessageHandler::ShareImages** method calls a static method on the **ShareDialog** class to show a modal dialog. The **ShareDialog** class will be covered in Chapter 15.

Touch Screen Gestures

Windows 7 provides support for gestures on touch screen computers. Gestures [http://msdn.microsoft.com/en-us/library/dd940543(v=VS.85).aspx] are movements of one or more fingers on the touch screen. Some gestures have corresponding mouse movements but since a multitouch screen can respond to two finger movements there are some gestures that cannot be replicated with the mouse.

Touch screen gestures are relayed to an application through the **WM_GESTURE** [http://msdn.microsoft.com/en-us/library/dd353242(v=VS.85).aspx] message. The **lParam** parameter of this message is a handle that is passed to the **GetGestureInfo** [http://msdn.microsoft.com/en-us/library/dd353235(v=VS.85).aspx] function to return information about the gesture in a **GESTUREINFO** [http://msdn.microsoft.com/en-us/library/dd353232(v=VS.85).aspx] structure. The caller of the **GetGestureInfo** function allocates the **GESTUREINFO** structure and the function fills the structure. Any code that handles the **WM_GESTURE** message must close the handle by calling the **CloseGestureInfoHandle** [http://msdn.microsoft.com/en-us/library/dd353228(v=VS.85).aspx] function. The members of the **GESTUREINFO** structure that you can use are shown in the following table.

Member	Description
cbSize	The size of the structure, in bytes.
dwFlags	The state of the gesture such as begin, inertia, and end.
dwID	An identifier to indicate the gesture that is happening.
ptsLocation	A **POINTS** structure containing the coordinates associated with the gesture. These coordinates are always relative to the origin of the screen.
ullArguments	A 64-bit unsigned integer that contains the arguments for gestures that fit into eight bytes. This is the extra information that is unique for each gesture type and is also passed through the message **wParam** parameter.

The type of gesture is identified through the **dwID** field of the **GESTUREINFO** structure (these values are shown in the following table) and the status of the gesture (whether the gesture has started or finished) is passed through the **wFlags** field.

Name	Description	ullArgument	ptsLocation
GID_ZOOM	The zoom gesture.	The distance between the two points.	The center of the zoom.
GID_PAN	The pan gesture.	The distance between the two points.	The current position of the pan.
GID_ROTATE	The rotation gesture.	The angle of rotation if the **GF_BEGIN** flag is set. Otherwise, the angle change since the rotation has started.	The center of the rotation.
GID_TWOFINGERTAP	The two-finger tap gesture.	The distance between the two fingers.	The center of the two fingers.
GID_PRESSANDTAP	The press and tap gesture.	The delta between the first finger and the second finger. This value is stored in the lower 32 bits of the **ullArgument** in a **POINT** structure.	The position that the first finger comes down on.

Hilo provides processing for just two of these gestures: **GID_PAN** and **GID_ZOOM**. To do this there are two virtual methods **OnPan** and **OnZoom** on the **WindowMessageHandler** class that is the base class of the message handler class hierarchy. The **WindowMessageHandler::OnMessageReceived** method handles messages sent to a window and Listing 4 shows how the message is handled. At the start of the code you can see where the **GetGestureInfo** function is called to obtain information about the gesture and at the end is the call to the **CloseGestureInfoHandle** function to clean up the resources if the message is handled.

Once the gesture information has been obtained, the handler code tests for the two gestures that can be handled. The handler then decodes the parameters appropriately before calling the virtual method to allow the child window message handler class to respond to the gesture.

Listing 4 Code to handle gesture messages

C++

```
case WM_GESTURE:
   {
      bool handled = false;

      GESTUREINFO info;
      info.cbSize = sizeof(info);
      if (::GetGestureInfo((HGESTUREINFO)lParam, &info))
      {
         switch(info.dwID)
         {
         case GID_PAN:
            {
               D2D1_POINT_2F panLocation =
Direct2DUtility::GetPositionForCurrentDPI(info.ptsLocation);
               hr = OnPan(panLocation, info.dwFlags);
               if (SUCCEEDED(hr))
               {
                  if (S_OK == hr)
                  {
                     handled = true;
```

```
                    }
                }
                break;
            }

        case GID_ZOOM:
            {
                static double previousValue = 1;
                switch(info.dwFlags)
                {
                case GF_BEGIN:
                      hr = OnZoom(1.0f);
                      break;
                case 0:
                      hr = OnZoom(static_cast<float>(LODWORD(info.ullArguments) /
previousValue));
                      break;
                }

                if (SUCCEEDED(hr))
                {
                   previousValue = LODWORD(info.ullArguments);
                   if (S_OK == hr)
                   {
                      handled = true;
                   }
                }

                break;
            }
        }
    }

    if (handled)
    {
        ::CloseGestureInfoHandle((HGESTUREINFO)lParam);
        *result = 0;
    }
    else
    {
        *result = 1;
    }

    break;
  }
```

Implementing the Pan Gesture

The pan gesture occurs when you sweep your finger across the touch screen. Typically an application will animate an item on screen to mirror this movement. Windows 7 provides inertia notifications so that an application can respond accordingly. After you take your finger off the touch screen at the end of a pan gesture, Windows 7 calculates the trajectory based on the velocity and angle of motion. It continues to send **WM_GESTURE** messages of type **GID_PAN** that are flagged with **GF_INERTIA**. Windows 7 will send these messages reducing the speed of the movement so that eventually the gesture messages stop. In effect the Windows 7 inertia engine is providing the positions for a deceleration animation.

The Browser application handles touch gestures in the media pane. The pan gesture is used to move between photos. It is handled by the **MediaPaneMessageHandler::PanImage** method. This method will only scroll one photo position for each individual pan gesture. To scroll further you have to repeat the gesture. The **PanImage** method handles the start of the pan gesture by storing the start position. When the next message for the gesture is received, the **PanImage** method can use the previous position to determine by how much the photo should

pan.

The Browser interprets a pan gesture to mean, move to the next photo in the following direction. Once the gesture has started, the Browser will always complete the action even if you change the finger movement. For example, if you pan to the left and then before removing your finger pan to the right, Browser will always pan to the left photo.

The panning movement is carried out through an animation and will occur after the gesture has finished. Since the Browser will only pan one photo position at a time, it will not respond to the inertia messages for the gesture, in fact it treats them as indicating that the gesture has ended. The reason is that the Windows 7 inertia engine calculates the inertia response according to the speed of the finger movement and it could result in scrolling several photo positions. When the message is received indicating that the pan gesture has ended (or is generating inertia messages) the media pane completes the scrolling of the photos by rendering an acceleration-deceleration animation with a decelerating movement from the current position to the final position.

Implementing the Zoom Gesture

The zoom gesture is a pinch between two fingers: if the distance between the fingers decreases the indication is that the item size should decrease (zoom out) and if the distance between the fingers increases the indication is that the item size should increase (zoom in).

Listing 4 shows that when the zoom gesture starts, the **dwFlags** parameter is **GF_BEGIN** and this is handled by saving the current distance between the two fingers. This value is used when handling subsequent zoom gesture messages to determine the change in the size of the image. The subsequent messages as part of this gesture have a value of 0 for the **dwFlags** parameter. These subsequent messages are handled by zooming to the proportional change in the distance between the fingers since the last message.

Conclusions

The Hilo Browser provides several UI features. This chapter explained how the Browser implements the buttons that allow you to launch the Annotator and Share applications, and it explained how the touch screen gestures were implemented. In the next chapter we will see how the Hilo applications provide jump lists and taskbar thumbnail images.

Chapter 14: Adding Support for Windows 7 Jump Lists & Taskbar Tabs

The Hilo Browser and Annotator support Windows 7 Jump Lists and taskbar tabs. Jump Lists provide the user with easy access to recent files and provide a mechanism to launch key tasks. Taskbar tabs provide a preview image and access to additional actions within the Windows taskbar. In this Chapter we will see how the Hilo Browser and Annotator applications implement support for Windows 7 Jump Lists and taskbar tabs.

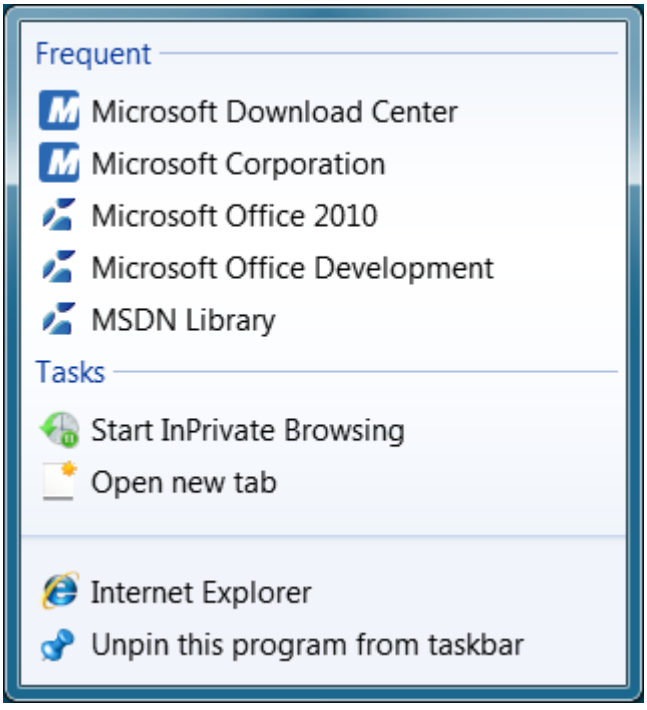
Implementing Jump List Items

The Jump List for an application is displayed when you right-click on the taskbar tab for the application. On a touch screen you can show the Jump List in three ways: touch the tab with one finger and then touch the required screen with another finger, use the pan gesture dragging the tab away from the taskbar, or touch the tab until a circle appears and then let go.

The Jump List contains two types of items, destinations and tasks. For example, Figure 1 shows the Jump List for Internet Explorer, there are two categories: **Frequent** and **Tasks**. The **Frequent** list items are web pages that are frequently displayed and since these refer to content they are described as destinations. Other applications will have files, folders, or some other content-based items as destinations and typically these are items that can be represented by an **IShellItem** [http://msdn.microsoft.com/en-us/library/dd378422(VS.85).aspx] object (or sometimes an **IShellLink** [http://msdn.microsoft.com/en-us/library/bb774950(VS.85).aspx] object). The screenshot shows five items in the **Frequent** list and you can set a limit to how many items are shown through the **Customize Start Menu** dialog accessed through the **Start** menu properties.

The **Tasks** list items are things that you can do with Internet Explorer: start private browsing and open a new tab. Since tasks need to have information about a process that will perform the task and parameters indicating the task action, tasks are **IShellLink** [http://msdn.microsoft.com/en-us/library/ms646927(VS.85).aspx] objects.

Figure 1 Jump List for Internet Explorer



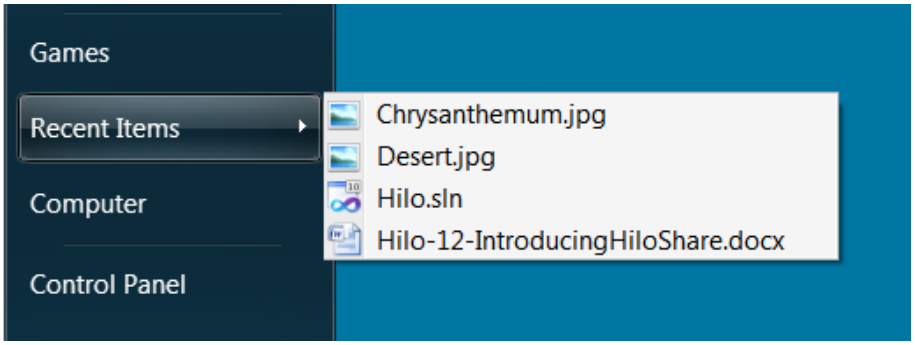
The Windows 7 Shell API allows you to alter the Jump List in several ways. The simplest action is to add a destination to the **Recent** file list because in many cases Windows 7 will do this without any code, although you

can specifically add a destination to this list. The Shell API provides access through the **ICustomDestinationList** [http://msdn.microsoft.com/en-us/library/dd378395(v=VS.85).aspx] interface to allow you to customize the Jump List to add custom categories and tasks.

Recent File Lists

Recent file lists have been a feature of Windows since Windows XP and are provided through the **SHAddToRecentDocs** [http://msdn.microsoft.com/en-us/library/dd378399(v=VS.85).aspx] Windows 7 Application Programming Interface (API) function. This function adds a document to the **My Recent Documents** (Windows XP) or **Recent Items** (Windows Vista and later) menu on the **Start** menu. Figure 2 shows an example of the **Recent Items** menu where there is Word document, a Visual Studio solution, and two image files. The entries in the **Recent Items** menu are paths to data files and the shell uses file association registry information to determine which application opens the file.

Figure 2 Showing the Windows 7 Recent Items menu



Windows 7 extends the concept of the **Recent Items** list to individual applications so that the **SHAddToRecentDocs** function not only adds a file to the system's **Recent Items** menu but also adds the file to the application's **Recent** Jump List. Figure 3 shows the Jump List for Annotator, the lower three items are tasks added to every Jump List and allow you to start a new instance of the application, pin the application item to the taskbar, and to close down an instance. The top items are the **Recent** file list and are recent files opened with Annotator. Items are added to the **Recent** list if your application opens a file with one of the Shell file APIs (the **GetOpenFileName** [http://msdn.microsoft.com/en-us/library/aa380072(v=VS.85).aspx] function or the **IFileOpenDialog** [http://msdn.microsoft.com/en-us/library/dd378402(VS.85).aspx] interface) or if your application explicitly calls the **SHAddToRecentDocs** function.

Figure 3 Showing the Jump List for Annotator

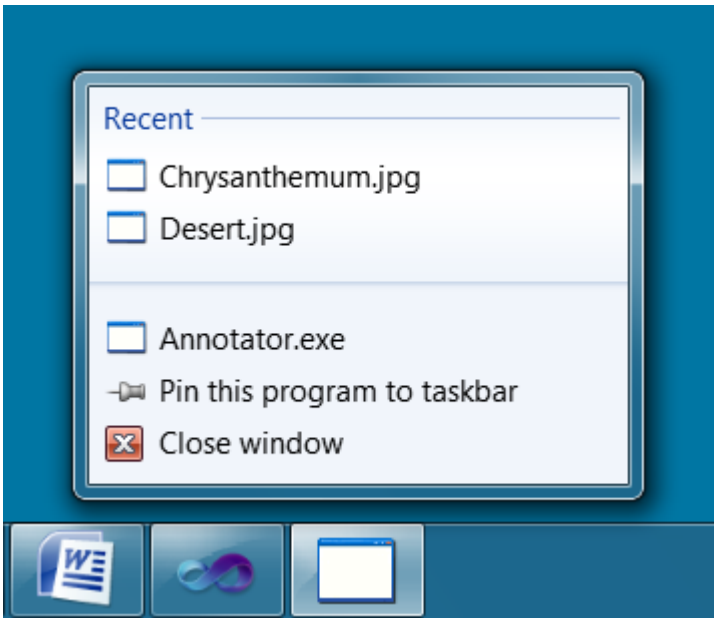


Figure 3 shows that two image files have been opened recently with Annotator and Figure 2 shows that these files

are also displayed on the system-wide Windows 7 **Recent Items** menu. Since the **SHAddToRecentDocs** function adds files to the **Recent Items** list this raises the question of how Windows 7 knows that these files were opened by Annotator and not some other image editor. The answer lies in application user model IDs (or AppID for short).

AppIDs are strings that identify an application and allow the Windows 7 shell to stack the tabs on the taskbar for multiple instances of the same application. Windows 7 can allocate an AppID for an application, but you can specify the string yourself which gives you greater flexibility. The API function to do this is **SetCurrentProcessExplicitAppUserModelID** [http://msdn.microsoft.com/en-us/library/bb761474(VS.85).aspx] and holds the current record for being the Windows 7 API function with the longest name. This function takes a single parameter which is the AppID as a Unicode string. AppIDs can be any unique string but the recommendation is to use the same format as ProgID strings, although clearly you should use a different value to your application's ProgID. Listing 1 shows the declaration of constant strings at the top of the AnnotatorApplication.cpp file which defines the ProgID and AppID strings.

Listing 1 Declaration of the ProgID and AppID for Annotator

C++

```
const std::wstring ProgID = L"Microsoft.Hilo.AnnotatorProgID";
const std::wstring FriendlyName = L"Microsoft Hilo Annotator";
const std::wstring AppUserModelID = L"Microsoft.Hilo.Annotator";
const std::wstring FileTypeExtensions[] =
{
    L".bmp",
    L".dib",
    L".jpg",
    L".jpeg",
    L".jpe",
    L".jfif",
    L".gif",
    L".tif",
    L".tiff",
    L".png"
};
```

The **SetCurrentProcessExplicitAppUserModelID** function must be called before the first window is created and in Annotator this function is called in the **AnnotatorApplication::Initialize** method before the main window is created. This function associates the specified AppID with the application and its windows but it makes no connection between the application and the files that it can edit. That is the role of the ProgID registry setting. Listing 2 shows part of the registration that is required for Annotator to add files to the **Recent** file Jump List. The first few lines shows that Annotator has to be registered as a handler for the file types that it edits and Listing 2 only gives the registration for the .bmp file type (all the file types mentioned in the **FileTypeExtensions** array in Listing 1 will have similar values). The important point is that the file type has a string entry in its **OpenWithProgIDs** [http://msdn.microsoft.com/en-us/library/bb166549(VS.80).aspx] key indicating that the file can be opened with Annotator. Note that Annotator will not be the only application registered as a handler for the file type.

Listing 2 Registration for Jump Lists

C++

```
[HKEY_CLASSES_ROOT\.bmp\OpenWithProgids]
"Microsoft.Hilo.AnnotatorProgID"="":

[HKEY_CLASSES_ROOT\Microsoft.Hilo.AnnotatorProgID]
"FriendlyTypeName"="Microsoft Hilo Annotator"
"AppUserModelID"="Microsoft.Hilo.Annotator"

[HKEY_CLASSES_ROOT\Microsoft.Hilo.AnnotatorProgID\CurVer]
@="Microsoft.Hilo.AnnotatorProgID"

[HKEY_CLASSES_ROOT\Microsoft.Hilo.AnnotatorProgID\Shell\Open\Command]
```

```
@="C:\\Hilo\\annotator.exe %1"
```

The ProgID registration contains a value called **AppUserModelID** [http://msdn.microsoft.com/en-us/library/bb775834(VS.85).aspx] that has the AppID for the application. This registration value links the AppID to the ProgID. For the shell to be able to launch an instance of the application there must be information about the location and command line format for the application. This is the purpose of the **Shell\Open\Command** subkey. The default value of this key contains the path and an indication that the data file should be the first command line parameter. Chapter 12 [http://msdn.microsoft.com/en-us/library/gg241211.aspx] described the **RegistrationHelper** utility that is called automatically when the Hilo Annotator application is first used. This utility does the registration to enable Annotator to use Jump Lists.

There are three ways to load a file in Annotator:

- **Through the command line.** When you click on the **Annotator** button in the Hilo Browser it starts the Annotator with the full path to the currently selected photo.
- **Through selecting another file in the Annotator image editor.** When you start Annotator without a filename, the application shows all the photos in the Pictures library and you can scroll through the list of pictures for the photo you want to edit.
- **Through the Open menu item on the main menu of Annotator.** When you click the **Open** menu item it shows the standard **File Open** dialog through which you can browse for one or more files to edit.

The last option in this list will use the standard **File Open** dialog through the **IFileOpenDialog** interface and when you open a file with this API the shell will automatically make a call to the **SHAddToRecentDocs** function. The other two methods to open a file do not use a Windows 7 API that adds the file to the Jump List and so Annotator must do this explicitly with a call the **SHAddToRecentDocs**. This call is made in the **ImageEditorHandler::SaveFileAtIndex** method, which is called when you click the **Save** or **Save A Copy As** menu items in Annotator and also when you close the application.

Jump List Tasks

The **Recent** file list is just one way that you can add items to a Jump List. The Windows 7 shell allows you to customize Jump Lists further, and Browser illustrates this by adding a custom task. The key to customizing the Jump List is the **ICustomDestinationList** [http://msdn.microsoft.com/en-us/library/bb762105(VS.85).aspx] interface. Changes to a Jump List are done in a transactional way: you start by calling the **ICustomDestinationList::BeginList** [http://msdn.microsoft.com/en-us/library/dd378398(v=VS.85).aspx] method, make the changes, and then call the **ICustomDestinationList::CommitList** [http://msdn.microsoft.com/en-us/library/dd391703(v=VS.85).aspx] method to make the changes permanent. If the application has an explicit AppID then you must call the **ICustomDestinationList::SetAppID** [http://msdn.microsoft.com/en-us/library/dd378307(VS.85).aspx] method before calling the **BeginList** method.

Browser does this work in the **JumpList** class which is called at the end of the **BrowserApplication::Initialize** method when the application is first created. Listing 3 shows the code to add a custom task with the **JumpList** class. Browser assumes that Annotator is in the same folder and so the first action is to get the path for this folder and use it to create a full path to Annotator. Next an instance of the **JumpList** class is created and initialized with the AppID of the application. Finally, the **JumpList::AddUserTask** method is called to create a task that will launch the Annotator application with no command line parameters.

Listing 3 Creating a task in the Browser Jump List

C++

```
wchar_t currentFileName[FILENAME_MAX];

if (!GetModuleFileName(nullptr, currentFileName, FILENAME_MAX))
{
   return S_OK;
}

// Annotator should be found in the same directory as this binary
std::wstring currentDirectory = currentFileName;
std::wstring externalFileName = currentDirectory.substr(0, currentDirectory.find_last_of(L"\\")
```

```
+ 1);
externalFileName += L"annotator.exe";

JumpList jumpList(AppUserModeId);
hr = jumpList.AddUserTask(externalFileName.c_str(), L"Launch Annotator", nullptr);
```

Listing 4 shows the main code in the **JumpList::AddUserTask** method. First, this code creates the destination list object and identifies the Jump List to alter through a call to the **ICustomDestinationList::SetAppID** method. Next the code indicates that it will start changing the Jump List by calling the **ICustomDestinationList::BeginList** method. This method returns two items, the first is the number of items that will be shown in the **Recent** or **Frequent** destination lists (set through the **Start** menu properties dialog) and the other object is a collection of items that you have remove from the Jump List (by right-clicking on an item and clicking **Remove from this list** item on the context menu). Browser does not process the removed item list and so although it stores the array as a member of the **JumpList** class it does nothing with this interface pointer. The code then calls the **JumpList::CreateUserTask** method to add the task item before calling the **ICustomDestinationList::CommitList** method to complete the changes.

Listing 4 Creating a new category for Browser

C++

```
HRESULT hr = CoCreateInstance(
   CLSID_DestinationList, nullptr, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&m_destinationList));

hr = m_destinationList->SetAppID(m_appId.c_str());
UINT cMinSlots;
hr = m_destinationList->BeginList(&cMinSlots, IID_PPV_ARGS(&m_objectArray));

hr = CreateUserTask(applicationPath, title, commandLine);
hr = m_destinationList->CommitList();
```

The **JumpList::CreateUserTask** method is shown in Listing 5. This code creates an object collection object (accessed through an **IObjectCollection** [http://msdn.microsoft.com/en-us/library/ms647591(VS.85).aspx] interface) that will contain the items to add to the Jump List. The code initializes this object by adding a shell link object to the task and then adding the collection to the user task list with a call to the **ICustomDestinationList::AddUserTasks** [http://msdn.microsoft.com/en-us/library/dd391704(v=VS.85).aspx] method. The user task list is a standard category called **Tasks** but you can create a custom category by calling the **ICustomDestinationList::AppendCategory** [http://msdn.microsoft.com/en-us/library/dd378396(v=VS.85).aspx] method and adding the category items to the return object array.

Listing 5 Creating a new category on Browser's Jump List

C++

```
ComPtr<IObjectCollection> shellObjectCollection;
HRESULT hr = CoCreateInstance(
   CLSID_EnumerableObjectCollection, nullptr, CLSCTX_INPROC,
IID_PPV_ARGS(&shellObjectCollection));

// Create shell link first
ComPtr<IShellLink> shellLink;
hr = CreateShellLink(applicationPath, title, commandLine, &shellLink);
hr = shellObjectCollection->AddObject(shellLink);

// Add the specified user task to the Task category of a Jump List
ComPtr<IObjectArray> userTask;
hr = shellObjectCollection->QueryInterface(&userTask);
hr = m_destinationList->AddUserTasks(userTask);
```

Tasks are typically shell link objects because this allows you to provide the path to an application rather than a document. In Browser the shell link object is created by a call to the **JumpList::CreateShellLink** method which is shown in Listing 6. This code creates and initializes a shell link object (**shellLink**) which is returned to the calling

method, **JumpList::CreateUserTask**. The shell link object has several properties which include the path of the application that will be executed and the command line arguments passed to the application. Most of the properties are set through **Set** methods on the **IShellLink** interface, but there is no such method for the title that will be displayed by the Jump List. So to set this value the **JumpList::CreateShellLink** method obtains the extended properties of the shell link object by querying for the **IPropertyStore** [http://msdn.microsoft.com/en-us/library/dd378459(VS.85).aspx] interface. The method then sets the **PKEY_Title** property with the task name. Properties are **PROPVARIANT** [http://msdn.microsoft.com/en-us/library/dd391701(v=VS.85).aspx] structures which are used in a similar fashion as **VARIANT** structures: they have to be initialized before use and then cleared when they are no longer needed.

Listing 6 Creating a shell link object

C++

```
ComPtr<IShellLink> shellLink;
HRESULT hr = CoCreateInstance(CLSID_ShellLink, nullptr, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(&shellLink));

// Set the path and file name of the shell link object
hr = shellLink->SetPath(applicationPath);

// Set the command-line arguments for the shell link object
hr = shellLink->SetArguments(commandLine);

// Set the name of the shell link object
ComPtr<IPropertyStore> propertyStore;
hr = shellLink->QueryInterface(&propertyStore);
PROPVARIANT propertyValue;
hr = InitPropVariantFromString(title, &propertyValue);
hr = propertyStore->SetValue(PKEY_Title, propertyValue);
hr = propertyStore->Commit();
hr = shellLink->QueryInterface(shellLinkAddress);
PropVariantClear(&propertyValue);
```

Implementing the Annotator Taskbar Tab

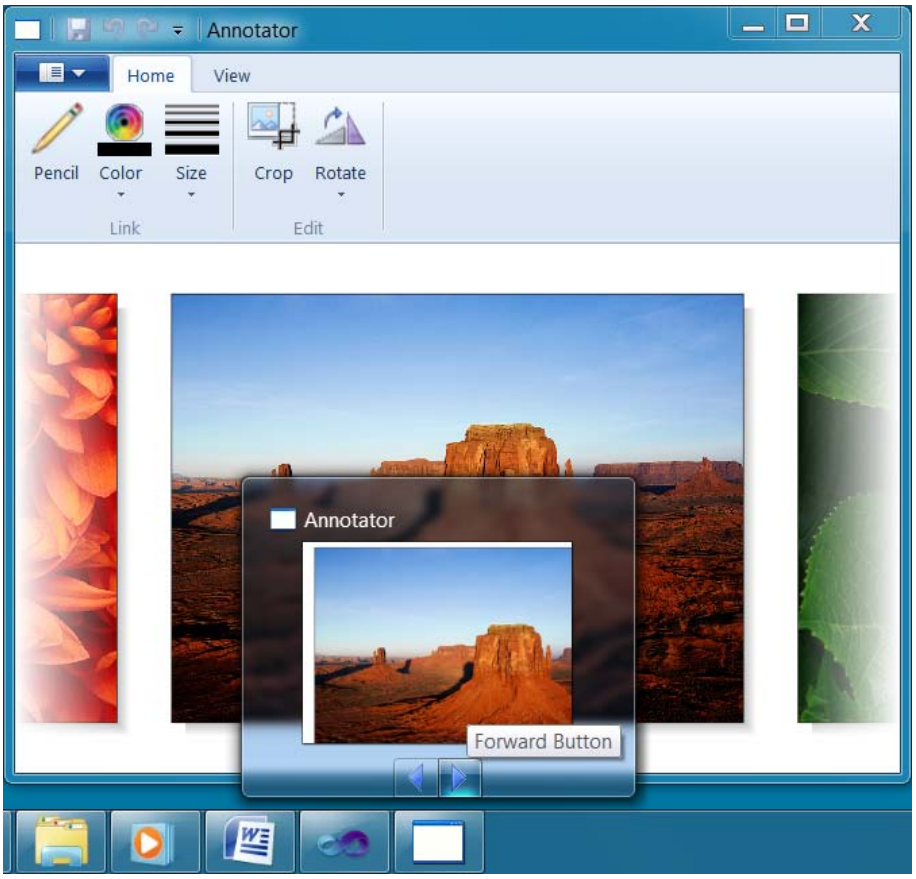
Windows 7 provide thumbnails for running applications through their taskbar tabs. When you hover the mouse over a tab, or with a touch screen you touch the tab and hold your finger on the tab for until the thumbnail appears. By default this thumbnail will be a preview of the entire window as shown in Figure 4 for Browser.

Figure 4 Showing the taskbar thumbnail for Browser

The thumbnail is a portion of the main window of the application and by default the entire window is shown. but your code can provide a specific part of the window. In addition, you can add controls to the thumbnail window so that you can have basic access to the application even when the application is minimized.

Annotator provides code to change the section of the window that is shown in the thumbnail and to add controls to allow you to scroll the images in the image editor. To see this, start Annotator on its own, either through the **Start** menu (type **Annotator.exe** in the **Start** menu search box and click the link returned) or through the **Launch Annotator** task on the Browser Jump List. In both cases Annotator will be started with more than one image in the image editor pane. At this point hover the mouse cursor over the taskbar tab for Annotator and you will see that the thumbnail for Annotator only has the image currently being edited, as shown in Figure 5. The thumbnail does not show the ribbon, and in addition, there are two buttons at the bottom of the thumbnail called **Backward Button** and **Forward Button**. When you click on either of these buttons the image in Annotator will change as if you pressed the keyboard left or right arrow keys.

Figure 5 Showing the Taskbar thumbnail for Annotator



When you change the size of the image in the Annotator by using the zoom buttons, the image shown in the thumbnail is zoomed appropriately.

The code that does most of the work is in a class called **Taskbar**. This class wraps access to the task bar list object that implements the **ITaskbarList3** [http://msdn.microsoft.com/en-us/library/dd378403(v=VS.85).aspx] interface. The thumbnail shows the contents of a window and allows you to add child controls to the thumbnail. These controls will send command messages to a window and therefore many of the **ITaskbarList3** interface methods need a **HWND** parameter. The **Taskbar** class constructor takes a **HWND** parameter which is stored as a member variable and this handle is passed to the object methods that need a window handle. The other important member variable is the interface pointer to the task bar list object and this is initialized by the **Initialize** method, which is called by all the public methods. The **Taskbar::Initialize** method is shown in Listing 7, this simply creates the task bar list object and initializes it.

Listing 7 Creating the task bar list object

C++

```
HRESULT Taskbar::Initialize()
{
   HRESULT hr = S_OK;
   if (!m_taskbarList)
   {
      hr = CoCreateInstance(
         CLSID_TaskbarList, nullptr, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&m_taskbarList));
      if (SUCCEEDED(hr))
      {
         hr = m_taskbarList->HrInit();
      }
   }
   return hr;
}
```

Thumbnail Image

Providing the thumbnail image is relatively straightforward: all you need to do is give the taskbar list object a rectangle that contains coordinates for the window's client area. Once you have done this, the taskbar will obtain the appropriate part of the window when necessary and draw it in the thumbnail. You do not have to do any additional drawing.

In Annotator a request to redraw the application window is routed through to the **ImageEditorHandler::OnRender** method which calls the **ImageEditorHandler::UpdateTaskbarThumbnail** method. The majority of this method calculates the clipping rectangle for the current photo (in Listing 8 this rectangle is the **rect** variable). The code that sets the current clipping rectangle of the window to show in the thumbnail is shown in Listing 8. The **Taskbar::SetThumbnailClip** method simply calls the **ITaskbarList3::SetThumbnailClip** [http://msdn.microsoft.com/en-us/library/dd378402(VS.85).aspx] method passing the windows handle passed to the **Taskbar** class constructor (**hWndParent** the handle of the main window) and the clipping rectangle.

Listing 8 Providing the thumbnail clip rectangle

C++

```
static Taskbar taskbar(hWndParent);
// Zoom the image to the thumbnail
hr = taskbar.SetThumbnailClip(&rect);
```

Thumbnail Buttons

There are several steps needed to add a button to a thumbnail. First you need to indicate to the taskbar the images that will be used for the buttons, then you need to add the buttons and provide information like tooltip captions and an ID, and finally you have to provide code to handle the command messages generated when a button is clicked. In Annotator the **Taskbar** class does the work of adding buttons and their images to the taskbar and this code is called in the **AnnotatorApplication::Initialize** method, the code is shown in Listing 9. The **Taskbar** object is initialized with the handle of the main window of the Annotator application. This is used to indicate to the taskbar that the buttons will be shown for thumbnails for this particular window. The **ThumbnailToolbarButton** structure shown in this code is used to hold the identifier for the button and an indication as to whether the button is enabled (at this point, by default, both buttons are enabled). When you click on the button a **WM_COMMAND** [http://msdn.microsoft.com/en-us/library/bb774950(VS.85).aspx] message is sent to the handler window and the identifier of the button will be provided as part of the **wParam** of this message.

Listing 9 Adding buttons to the thumbnail image

C++

```
HWND hwnd;
mainWindow->GetWindowHandle(&hwnd);
Taskbar taskbar(hwnd);
ThumbnailToobarButton backButton = {APPCOMMAND_BROWSER_BACKWARD, true};
ThumbnailToobarButton nextButton = {APPCOMMAND_BROWSER_FORWARD, true};
hr = taskbar.CreateThumbnailToolbarButtons(backButton, nextButton);
```

The **Taskbar::CreateThumbnailToolbarButtons** method does the main work of adding the buttons to the thumbnail. The thumbnail button images are provided through a standard Windows 7 image list control and the first action of the **CreateThumbnailToolbarButtons** method is to initialize the task bar list object with an image list and the code to do this is provided by the **Taskbar::SetThumbnailToolbarImage** method shown in Listing 10. This code is straightforward; the resources of the Annotator contain two images arrow_toolbar_16.bmp and arrow_toolbar_24.bmp which are used when the system uses, respectively, 16x16 and 24x24 pixels icons. Each image contains two button images and the computer's icon size setting determines which of these is used to initialize the **imageList** control through a call to the **ImageList_LoadImage** [http://msdn.microsoft.com/en-us/library/bb761557(VS.85).aspx] function. Once the image list has been initialized it is associated with the taskbar thumbnail for Annotator with a call to the **ITaskbarList3::ThumbBarSetImageList** [http://msdn.microsoft.com/en-us/library/dd391692(v=VS.85).aspx] method.

Listing 10 Setting the thumbnail image

C++

```
// Get the recommended width of a small icon in pixels
int const smallIconWidth = GetSystemMetrics(SM_CXSMICON);
// Load the bitmap based on the system's small icon width
HIMAGELIST imageList;
if (smallIconWidth <= 16)
{
   imageList = ImageList_LoadImage(
      HINST_THISCOMPONENT, MAKEINTRESOURCE(IDB_BITMAP_TOOLBAR_16),
      16, 0, RGB(255, 0, 255), IMAGE_BITMAP, LR_CREATEDIBSECTION);
}
else
{
   imageList = ImageList_LoadImage(
      INST_THISCOMPONENT, MAKEINTRESOURCE(IDB_BITMAP_TOOLBAR_24),
      24, 0, RGB(255, 0, 255), IMAGE_BITMAP, LR_CREATEDIBSECTION);
}
// Add the tool bar buttons to the taskbar
if (imageList)
{
   hr = m_taskbarList->ThumbBarSetImageList(m_hWnd, imageList);
}
ImageList_Destroy(imageList);
```

Once the image list for the thumbnail controls has been loaded, the **Taskbar::CreateThumbnailToolbarButtons** method, Listing 11, can create the buttons. This method calls the **ITaskbarList3::ThumbBarAddButtons** [http://msdn.microsoft.com/en-us/library/dd391705(v=VS.85).aspx] method with an array that has information about the index of the button image in the image list, the tooltip string, and the ID of the control that will be passed through the **WM_COMMAND** message when the button is clicked.

Listing 11 Creating thumbnail buttons

C++

```
// Set the icon/images for thumbnail toolbar buttons
hr = SetThumbnailToolbarImage();
THUMBBUTTON buttons[2] = {};

// First button
buttons[0].dwMask = THB_BITMAP | THB_TOOLTIP | THB_FLAGS;
buttons[0].dwFlags = THBF_ENABLED | THBF_DISMISSONCLICK;;
buttons[0].iId = backButton.buttonId;
buttons[0].iBitmap = 0;
StringCchCopyW(buttons[0].szTip, ARRAYSIZE(buttons[0].szTip), L"Backward Button");

// Second button
buttons[1].dwMask = THB_BITMAP | THB_TOOLTIP | THB_FLAGS;
buttons[1].dwFlags = THBF_ENABLED | THBF_DISMISSONCLICK;
buttons[1].iId = nextButton.buttonId;
buttons[1].iBitmap = 1;
StringCchCopyW(buttons[1].szTip, ARRAYSIZE(buttons[1].szTip), L"Forward Button");

// Set the buttons to be the thumbnail toolbar
hr = m_taskbarList->ThumbBarAddButtons(m_hWnd, ARRAYSIZE(buttons), buttons);
```

Once the buttons have been created, you can change these values at a later stage by calling the **ITaskbarList3::ThumbBarUpdateButtons** [http://msdn.microsoft.com/en-us/library/bb761144(VS.85).aspx] method. Annotator does this in the **ImageEditorHandler::UpdateTaskbarThumbnail** method (which is called when the main window is updated). In this method there is code to test the position of the current photo in the list of photos in the image editor pane. If the current photo is the first photo in the list then the **Backward Button** is disabled and if the photo is at the end of the list the **Forward Button** is disabled. This action of enabling and disabling the thumbnail buttons is carried out by calling **Taskbar::EnableThumbnailToolbarButtons** and the relevant code is

shown in Listing 12. If a button should be enabled, this method uses the **THBF_ENABLED** flag and if the button is to be disabled the button uses the **THBF_DISABLED** flag.

Listing 12 Enabling thumbnail buttons

C++

```
HRESULT Taskbar::EnableThumbnailToolbarButtons(ThumbnailToobarButton backButton,
ThumbnailToobarButton nextButton)
{
    HRESULT hr = Initialize();
    if (SUCCEEDED(hr))
    {
        THUMBBUTTON buttons[2] = {};
        // First button
        buttons[0].dwMask = THB_BITMAP | THB_TOOLTIP | THB_FLAGS;
        if (backButton.enabled)
        {
            buttons[0].dwFlags = THBF_ENABLED | THBF_DISMISSONCLICK;
        }
        else
        {
            buttons[0].dwFlags = THBF_DISABLED;
        }
        buttons[0].iId = backButton.buttonId;
        buttons[0].iBitmap = 0;
        StringCchCopyW(buttons[0].szTip, ARRAYSIZE(buttons[0].szTip), L"Backward Button");
        // Second button
        buttons[1].dwMask = THB_BITMAP | THB_TOOLTIP | THB_FLAGS;
        if (nextButton.enabled)
        {
            buttons[1].dwFlags = THBF_ENABLED | THBF_DISMISSONCLICK;
        }
        else
        {
            buttons[1].dwFlags = THBF_DISABLED;
        }
        buttons[1].iId = nextButton.buttonId;
        buttons[1].iBitmap = 1;
        StringCchCopyW(buttons[1].szTip, ARRAYSIZE(buttons[1].szTip), L"Forward Button");
        // Update the buttons of the thumbnail toolbar
        hr = m_taskbarList->ThumbBarUpdateButtons(m_hWnd, ARRAYSIZE(buttons), buttons);
    }
    return hr;
}
```

When you click on the **Forward Button** or **Backward Button** the **WM_COMMAND** message is sent to the window associated with the thumbnail (in Annotator this is the main window). This message is routed to the image editor class and handled in the **ImageEditorHandler::OnCommand** method. This method calls the **ImageEditorHandler::NextImage** or **ImageEditorHandler::PreviousImage** method depending on which button is pressed. The code in Listing 12 provides the **THBF_DISMISSONCLICK** flag for enabled buttons to indicate that when you click on the thumbnail button, the thumbnail will disappear. If you omit this flag then the thumbnail will remain on screen and you will see the scrolling action performed within the thumbnail and the Annotator window.

Conclusions

In this chapter you have learned how to implement support for Windows 7 Jump Lists and a taskbar thumbnail for the application. In the next chapter you will see how to use the Windows 7 Web Services API to upload a photo to an online photo sharing application.

Chapter 15: Using Windows HTTP Services

The Hilo Browser application allows you to upload photos to the Flickr online photo sharing application. To do this, Hilo uses Windows HTTP Services. This chapter will explore how this library is used in the Hilo Browser to implement its photo sharing feature.

Flickr Share Dialog

When Hilo uploads a photo to Flickr, it makes several calls to the Flickr web server. These calls are made to authenticate the Hilo application (and obtain a session token called a frob); to authorize the access of the Hilo Flickr application to upload a photo to a Flickr account (and obtain an access token) and then to upload the photo. These calls are made across the network, and potentially they can take a noticeable amount of time. Hilo has to wait for responses from the Flickr web server in such a way that the user is kept informed. This is the purpose of the **Share** dialog.

The Hilo Browser's user interface provides a button, labeled **Share** (Figure 1). When you click on this button you will see the **Share** dialog (Figure 2), which is implemented by the **ShareDialog** class. This class has static methods and allows you to upload either the selected photo, or all photos in the current folder. The **Share** dialog has effectively three sets of controls reflecting your progress through the mechanism of uploading photos. The first set of controls is shown in Figure 2. When you click the **Upload** button a progress bar is shown under the radio buttons (Figure 3) to display the progress of the upload; and when the upload is complete all the initial controls are hidden except for the **Cancel** button which is relabeled **Close**, and the **View Photos** link control is displayed (Figure 4). The same class is used for all versions of this dialog.

Figure 1 The Hilo Browser Share button

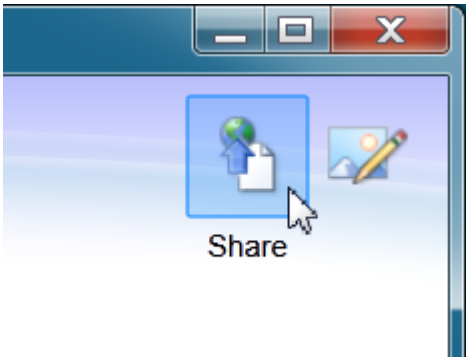


Figure 2 The initial display of the Share dialog

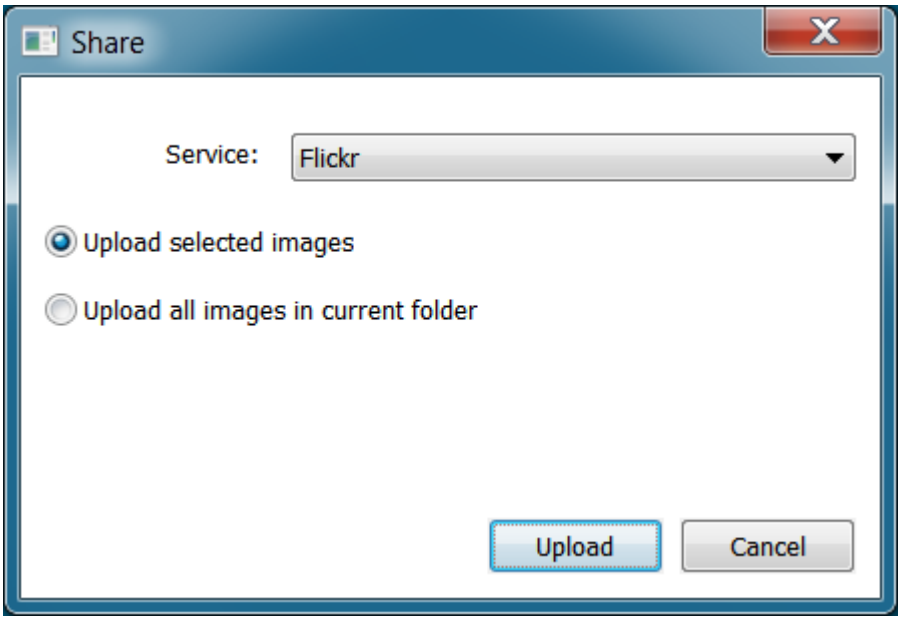


Figure 3 The display of the Share dialog when uploading photos

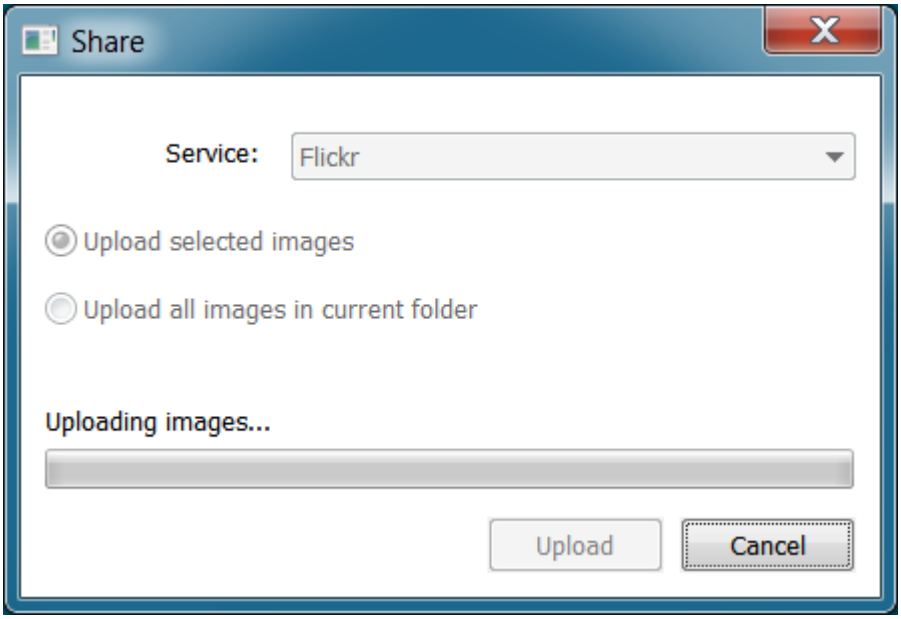
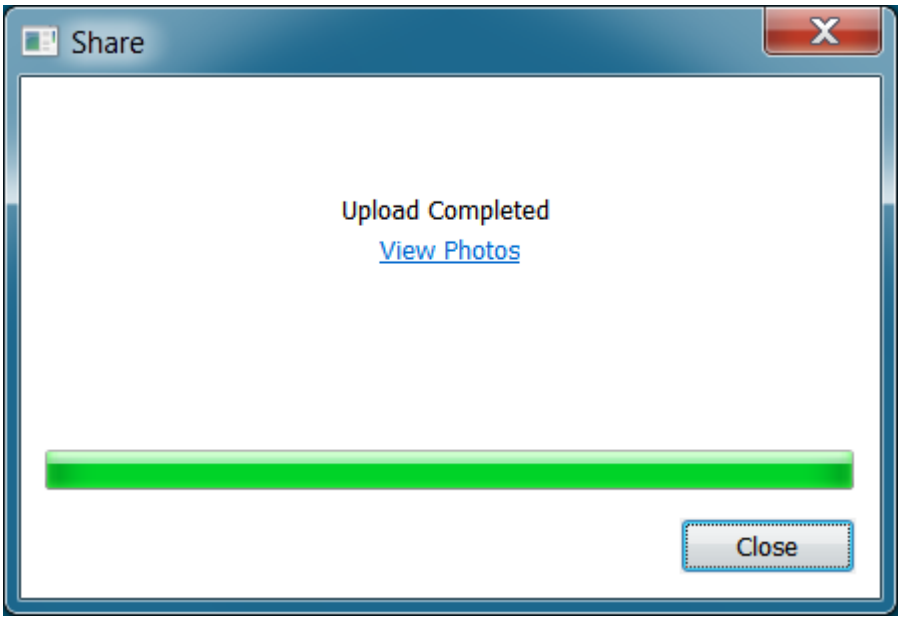


Figure 4 The display of the Share dialog when uploading has completed



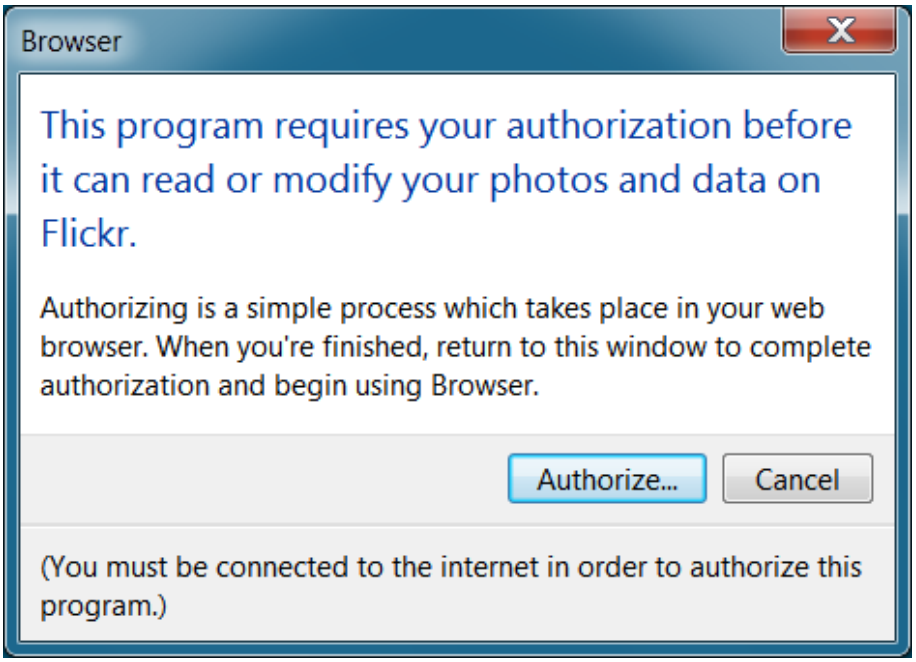
When the user clicks the **Upload** button, the **ShareDialog::UploadImages** method creates an instance of the **FlickrUploader** class to perform the network access. First, the **ShareDialog::UploadImages** method calls the **FlickrUploader::Connect** method to obtain the session key (the frob value) and launch the system registered browser to show the Flickr logon page. The user logs on to the Flickr account where Hilo will upload the photos. Next the **UploadImages** method calls the **FlickrUploader::GetToken** method to get the access token associated with the account where the photos will be uploaded. Finally, a new thread is created to run the **SharedDialog::ImageUploadThreadProc** method asynchronously. This method uploads each photo by calling the **FlickrUploader.UploadPhotos** method passing the token and the path to the photo. As each photo is uploaded the progress bar is updated by one position. When each photo is uploaded Flickr returns an ID and once all photos have been uploaded the **SharedDialog::ImageUploadThreadProc** method creates a URL to display the uploaded images. The format of this URL is shown in Listing 1 where the **ids** parameter is a comma separated list of the IDs of the photos to show. This URL is provided as the link of the **View Photos** link control in Figure 4.

Listing 1 Uploaded photos URL

```
http://www.flickr.com/photos/upload/edit/?ids=[comma separated list]
```

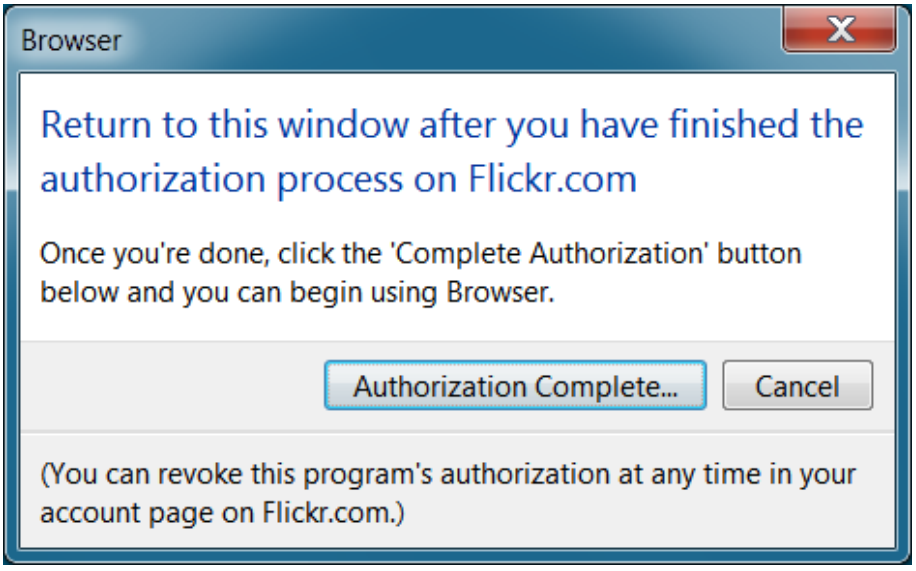
At several stages in the upload process the code has to pause for user input. The first time this happens is when the **ShareDialog::UploadImages** method obtains the frob from Flickr. The **UploadImages** method calls the **FlickrUploader::Connect** method which launches the system registered browser to show the Flickr logon page and the user task switches to this browser page. During this time the **UploadImages** method must be paused until the logon process has completed. To do this the **UploadImages** method displays two Task Dialog [http://msdn.microsoft.com/en-us/library/bb787471(v=VS.85).aspx] windows (created by calling the **TaskDialogIndirect** [http://msdn.microsoft.com/en-us/library/bb760544(v=VS.85).aspx] function).

Figure 5 Dialog informing the user that they must authorize Hilo



The first dialog (Figure 5) informs the user that they will have to authorize the Hilo application's access to their account. This lets the user know what is about to happen, and gives the user the opportunity to cancel the operation. When the user clicks the **Authorize** button on this first dialog, the **FlickrUploader::Connect** method is called to display the logon page and the second dialog (Figure 6) is shown in the background. This modal dialog blocks the **UploadImages** method and the user will see this dialog when they have finished the logon procedure and switched back to Browser. At this point the user can click the **Authorization Complete** button to unblock the **UploadImages** method and allow it to upload the photos by calling **SharedDialog:: ImageUploadThreadProc** method on a worker thread.

Figure 6 Dialog used to block Browser until the user has authorized the Hilo Flickr application



Uploading a Photograph

A photo can be several megabytes of data. The Flickr API for uploading photos involves a multi-part HTTP POST request to the **http://api.flickr.com/services/upload/** URI. Each part of the message is one of the arguments [http://www.flickr.com/services/api/upload.api.html] that describe the upload action. The action must be authenticated and authorized, and so the API key and the access token must be provided along with a message digest generated from them. The message digest (Flickr calls it the signature) is a hash generated from the method parameters and the secret known only by the Flickr application and Flickr. The digest is used by Flickr to determine the integrity of the parameters so that it can detect if the request has been altered en route, either

maliciously or by accident.

The final part of the message is the photo provided as binary data. Listing 2 shows the general format of the POST request (the items in square brackets will be replaced with actual data).

Listing 2 Example POST message to upload a photo

C++

```
POST /services/upload/ HTTP/1.1
Content-Type: multipart/form-data; boundary=--EBA799EB-D9A2-472B-AE86-568D4645707E
Host: api.flickr.com
Content-Length: [data_length]

--EBA799EB-D9A2-472B-AE86-568D4645707E
Content-Disposition: form-data; name="api_key"

[api_key_value]

--EBA799EB-D9A2-472B-AE86-568D4645707E
Content-Disposition: form-data; name="auth_token"

[token_value]

--EBA799EB-D9A2-472B-AE86-568D4645707E
Content-Disposition: form-data; name="api_sig"

[api_sig_value]

--EBA799EB-D9A2-472B-AE86-568D4645707E
Content-Disposition: form-data; name="photo"; filename="[filename]"

[image_binary_data]

--EBA799EB-D9A2-472B-AE86-568D4645707E
```

Using Windows HTTP Services

The network access to Flickr is performed by methods on the **FlickrUploader** class. This class makes two types of calls: calls to Web Services methods where the data is formatted and transmitted according to the SOAP protocol, and low level calls over HTTP to the Flickr website. The next chapter, Chapter 16, will describe the Windows 7 Web Services API code in Hilo. Windows HTTP Services provides access to web servers over the HTTP protocol and Hilo uses this API to upload photos to Flickr using the POST request shown in Listing 2. To use this API you have to include the Winhttp.h header file and link to the Winhttp.lib library.

Before you call any of the HTTP Services functions you must call the **WinHttpOpen** [http://msdn.microsoft.com/en-us/library/aa384098(VS.85).aspx] function. You pass to this function the name of the user agent that will be sent during future web requests, information about the proxy (if any) that will be used, and whether the web calls will be synchronous or asynchronous. The **WinHttpOpen** function returns a session handle (**HINTERNET**) that, like all the HTTP Services handles, must be closed with the **WinHttpCloseHandle** [http://msdn.microsoft.com/en-us/library/aa384090(v=VS.85).aspx] function when you have finished. Next you call the **WinHttpConnect** [http://msdn.microsoft.com/en-us/library/aa384091(v=VS.85).aspx] function to obtain a connection handle. For this function, you provide the session handle and the server name and port. This function does not make a network connection; it just prepares the internal connection settings. The next task is to create a request, to do this you call the **WinHttpOpenRequest** [http://msdn.microsoft.com/en-us/library/aa384099(v=VS.85).aspx] function, passing the connection handle and information about the request to be made. The **WinHttpOpenRequest** function returns a request handle which is used to store all the RFC822, MIME, and HTTP headers to be sent as part of the request. You add the actual headers for the request through a call to the **WinHttpAddRequestHeaders** [http://msdn.microsoft.com/en-us/library/aa384087(v=VS.85).aspx] function passing the request handle and a string that contains all the headers separated by return/linefeed pairs.

The actual network call is made when you call the **WinHttpSendRequest** [http://msdn.microsoft.com/en-

us/library/aa384110(v=VS.85).aspx] function passing the request handle. This function can be used to provide additional HTTP headers not provided by calls to **WinHtttpAddRequestHeaders** and any additional data required when the call is made for a PUT or POST request. The server will respond to the request, and you call the **WinHttpReceiveResponse** [http://msdn.microsoft.com/en-us/library/aa384105(v=VS.85).aspx] function with the request handle for the system to read the response headers from the server (which can be obtained by calling the **WinHttpQueryHeaders** [http://msdn.microsoft.com/en-us/library/aa384102(v=VS.85).aspx] function).

Once the server has responded you can receive data from the server. To do this you call the **WinHttpQueryDataAvailable** [http://msdn.microsoft.com/en-us/library/aa384101(v=VS.85).aspx] function with the request handle to receive the number of bytes that can be downloaded and then call the **WinHttpReadData** [http://msdn.microsoft.com/en-us/library/aa384104(v=VS.85).aspx] function to read the data. If the request is made synchronously, the **WinHttpQueryDataAvailable** function will block until there is data available.

Using HTTP Services in Hilo

Hilo uses HTTP Web Services when it uploads photos to Flickr. Listing 2 gives the general format of the POST request that is made to upload a photo and Listing 3 shows the **FlickrUploader::UploadPhotos** method that makes this call. The first few lines are straightforward. The code calls the **WinHttpOpen** function to indicate that the user agent is **Hilo/1.0** and that at this point no proxy will be specified. Next the code calls the **WinHttpConnect** function indicating that the host name is **api.flickr.com** and access will be made on the default HTTP port, the TCP port 80 and then calls the **WinHttpOpenRequest** function to specify that the request is a **POST** call to the **/Services/Upload/** resource.

Listing 3 Uploading photos to Flickr

C++

```
std::wstring FlickrUploader::UploadPhotos(
    const std::wstring& token, const std::wstring& fileName, bool* errorFound)
{
    std::wstring outputString;
    HINTERNET session = nullptr;
    HINTERNET connect = nullptr;
    HINTERNET request = nullptr;

    WINHTTP_AUTOPROXY OPTIONS  autoProxyOptions;
    WINHTTP_PROXY_INFO proxyInfo;
    unsigned long proxyInfoSize = sizeof(proxyInfo);
    ZeroMemory(&autoProxyOptions, sizeof(autoProxyOptions));
    ZeroMemory(&proxyInfo, sizeof(proxyInfo));

    // Create the WinHTTP session.
    session = ::WinHttpOpen(
        L"Hilo/1.0", WINHTTP_ACCESS_TYPE_NO_PROXY,
        WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
    connect = ::WinHttpConnect(session, L"api.flickr.com", INTERNET_DEFAULT_HTTP_PORT, 0);
    request = ::WinHttpOpenRequest(
        connect, L"POST", L"/services/upload/", L"HTTP/1.1",
        WINHTTP_NO_REFERER, WINHTTP_DEFAULT_ACCEPT_TYPES, 0);
    autoProxyOptions.dwFlags = WINHTTP_AUTOPROXY_AUTO_DETECT;
    // Use DHCP and DNS-based auto-detection.
    autoProxyOptions.dwAutoDetectFlags =
        WINHTTP_AUTO_DETECT_TYPE_DHCP | WINHTTP_AUTO_DETECT_TYPE_DNS_A;
    // If obtaining the PAC script requires NTLM/Negotiate authentication,
    // then automatically supply the client domain credentials.
    autoProxyOptions.fAutoLogonIfChallenged = true;

    if (FALSE != ::WinHttpGetProxyForUrl(
            session, L"http://api.flickr.com/services/upload/", &autoProxyOptions, &proxyInfo))
    {
        // A proxy configuration was found, set it on the request handle.
        ::WinHttpSetOption(request, WINHTTP_OPTION_PROXY, &proxyInfo, proxyInfoSize);
```

```
    }

    SendWebRequest(&request, token, fileName);
    outputString = GetPhotoId(&request, errorFound);

    // Clean up
    if (proxyInfo.lpszProxy)
    {
       GlobalFree(proxyInfo.lpszProxy);
    }
    if (proxyInfo.lpszProxyBypass)
    {
       GlobalFree(proxyInfo.lpszProxyBypass);
    }

    WinHttpCloseHandle(request);
    WinHttpCloseHandle(connect);
    WinHttpCloseHandle(session);
    return outputString;
}
```

The **FlickrUploader** class will upload the photo though a web call that will be made through a proxy if one is set up for the local network. To do this, the **UploadPhotos** method calls the **WinHttpGetProxyForUrl** [http://msdn.microsoft.com/en-us/library/aa384097(v=VS.85).aspx] function to receive the contents of the Proxy Auto-Configuration (PAC) file discovered using DHCP and DNS queries. The results are returned in the **proxyInfo** variable, which has a proxy server list and a proxy bypass list, that are allocated on the system global heap.These fields are freed with calls to the **GlobalFree** function when the method completes. The proxy information is associated with the request by calling the **WinHttpSetOption** [http://msdn.microsoft.com/en-us/library/aa384114(v=VS.85).aspx] method. At this point the method can make the web request by calling the **FlickrUploader::SendWebRequest** method shown in Listing 4.

The majority of the **SendWebRequest** method is to construct the headers of the HTTP POST request. The upload parameters have to be signed with the Flickr API secret and so the first part of the **SendWebRequest** method concatenates the secret, the API key, and the token. It then generates a MD5 hash of this string by calling the **FlickrUploader::CreateMD5Hash** method (which is described later). This header is added to the request by calling **WinHttpAddRequestHeaders**.

Listing 4 Making the web request

C++

```
int FlickrUploader::SendWebRequest(
    const HINTERNET *request, const std::wstring& token, const std::wstring& fileName)
{
    static const char* mimeBoundary = "EBA799EB-D9A2-472B-AE86-568D4645707E";
    static const wchar_t* contentType =
       L"Content-Type: multipart/form-data; boundary=EBA799EB-D9A2-472B-AE86-568D4645707E\r\n";

    // Parameters put in alphabetical order to generate the api_sig
    std::wstring params = flickr_secret;
    params += L"api_key";
    params += flickr_api_key;
    params += L"auth_token";
    params += token;

    std::wstring api_sig = CalculateMD5Hash(params);
    int result = ::WinHttpAddRequestHeaders(
       *request, contentType, (unsigned long)-1, WINHTTP_ADDREQ_FLAG_ADD);
    if (result)
    {
       std::wostringstream sb;
```

```
      sb << L"--" << mimeBoundary << L"\r\n";
      sb << L"Content-Disposition: form-data; name=\"api_key\"\r\n";
      sb << L"\r\n" << flickr_api_key << L"\r\n";

      sb << L"--" << mimeBoundary << L"\r\n";
      sb << L"Content-Disposition: form-data; name=\"auth_token\"\r\n";
      sb << L"\r\n" << token << L"\r\n";

      sb << L"--" << mimeBoundary << L"\r\n";
      sb << L"Content-Disposition: form-data; name=\"api_sig\"\r\n";
      sb << L"\r\n" << api_sig.c_str() << L"\r\n";

      sb << L"--" << mimeBoundary << L"\r\n";
      sb << L"Content-Disposition: form-data; name=\"photo\"; filename=\"" << fileName <<
L"\"\r\n\r\n";

      // Convert wstring to string
      std::wstring wideString = sb.str();
      int stringSize = WideCharToMultiByte(CP_ACP, 0, wideString.c_str(), -1, nullptr, 0,
nullptr, nullptr);
      char* temp = new char[stringSize];
      WideCharToMultiByte(CP_ACP, 0, wideString.c_str(), -1, temp, stringSize, nullptr,
nullptr);
      std::string str = temp;
      delete [] temp;

      // Add the photo to the stream
      std::ifstream f(fileName, std::ios::binary);
      std::ostringstream sb_ascii;
      sb_ascii << str;
      sb_ascii << f.rdbuf();
      sb_ascii << "\r\n--" << mimeBoundary << "\r\n";
      str = sb_ascii.str();
      result = WinHttpSendRequest(
          *request, WINHTTP_NO_ADDITIONAL_HEADERS,  0, (void*)str.c_str(),
          static_cast<unsigned long>(str.length()), static_cast<unsigned long>(str.length()),
0);
   }
   return result;
}
```

The remainder of the method adds the parameters to the request, a multi-part MIME message built as an ASCII string. First, the parts that indicate the API key, the token, and the signature are added using a Unicode buffer that is converted to an ASCII buffer before the contents of the photo are added as binary data. Finally the actual request is made to the server by calling the **WinHttpSendRequest** function.

Once the **FlickrUploader::UploadPhotos** method has made the request to the server it calls **FlickrUploader::GetPhotoId** to read the response from the server as shown in Listing 5. The first action is to call the **WinHttpReceiveResponse** method, which will block until the server sends the response. The **GetPhotoId** method then calls the **WinHttpQueryHeaders** function to get all the response headers. These headers are read in a single buffer because they are not needed, what is needed is the data that follows the response headers and these are obtained by calling the **WinHttpReadData** function.

Listing 5 Retrieving data from a web request

C++

```
std::wstring FlickrUploader::GetPhotoId(const HINTERNET *request, bool* errorFound)
{
   std::wstring outputString;
   int result = ::WinHttpReceiveResponse(*request, nullptr);
   unsigned long dwSize = sizeof(unsigned long);
```

```
    if (result)
    {
       wchar_t headers[1024];
       dwSize = ARRAYSIZE(headers) * sizeof(wchar_t);
       result = ::WinHttpQueryHeaders(
          *request, WINHTTP_QUERY_RAW_HEADERS, nullptr, headers, &dwSize, nullptr);
    }
    if (result)
    {
       char resultText[1024] = {0};
       unsigned long bytesRead;
       dwSize = ARRAYSIZE(resultText) * sizeof(char);
       result =::WinHttpReadData(*request, resultText, dwSize, &bytesRead);
       if (result)
       {
          // Convert string to wstring
          int wideSize = MultiByteToWideChar(CP_UTF8, 0, resultText, -1, 0, 0);
          wchar_t* wideString = new wchar_t[wideSize];
          result = MultiByteToWideChar(CP_UTF8, 0, resultText, -1, wideString, wideSize);
          if (result)
          {
             std::wstring photoId = GetXmlElementValueByName(wideString, L"photoid",
errorFound);
             if (!(*errorFound))
             {
                outputString = photoId;
             }
          }
          delete [] wideString;
       }
    }
    return outputString;
}
```

The response from the server will be an XML string. The **FlickrUploader::GetXmlElementValueByName** method uses the XMLLite API to create an XML reader object by calling the **CreateXmlReader** [http://msdn.microsoft.com/en-us/library/ms752822(VS.85).aspx] function and then iterating over every element until it finds the **photoid** element and returns the value of the element as the ID of the photo just uploaded. The **SharedDialog::ImageUploadThreadProc** method stores these IDs, and then when all photos have been uploaded it creates a URL to the page that display the uploaded images.

Using Cryptographic API: Next Generation

Whenever you make a call to a Flickr service method you must pass a message digest along with the method parameters. The message digest allows Flickr to determine the integrity of the parameters so that it can detect if the request has been altered during transmission, either maliciously or by accident. The digest is generated from a string that is the concatenation of the Hilo Flickr application secret, the named parameters in alphabetic order and the Web Service method name. The digest is the hex string value of the MD5 hash of this parameter string.

The digest is not used for privacy, indeed, the parameters are passed cleartext along with the message digest. When Flickr receives a request it obtains the secret for the Flickr client application (the Hilo Browser)making the request and using this and the parameters it can also create an MD5 hash. Flickr compares the hash it created with the message digest of the request, and if the two are the same then it indicates that the message has not been tampered with or corrupted en route.

To create a MD5 hash Hilo uses the Windows Cryptography API: Next Generation (CNG) library that was introduced with Windows Vista. Earlier versions of Windows provided cryptographic functions through the CryptoAPI. CNG is much improved compared to CryptoAPI: it provides more cryptographic providers, including the newer algorithms that are part of the National Security Agency (NSA) Suite B, and the API is more logically factored. All features of CNG are accessed using the same steps: open the provider, get or set properties of the algorithm, perform the cryptographic action and then close the provider.

To open a provider you call the **BCryptOpenAlgorithmProvider** [http://msdn.microsoft.com/en-us/library/aa375479(v=VS.85).aspx] function and provide the names of the algorithm, the provider, and any appropriate flags. The function returns a **BCRYPT_ALG_HANDLE** handle, which you can use to get or set the properties of the algorithm. When you have finished with the provider you must close it by passing the handle to the **BCryptCloseAlgorithmProvider** [http://msdn.microsoft.com/en-us/library/aa375377(v=VS.85).aspx] function.

You can alter how a cryptographic algorithm works, or obtain information about the algorithm through properties. To obtain a property you call the **BCryptGetProperty** [http://msdn.microsoft.com/en-us/library/aa375464(v=VS.85).aspx] function, passing the handle for an open provider, the name of the property [http://msdn.microsoft.com/en-us/library/aa376211(v=VS.85).aspx] , and a pointer to a caller allocated buffer to receive the property value. The function also has an in/out parameter for the size of the buffer. Some properties are of a known size (for example **BCRYPT_KEY_LENGTH** which returns the size of the cryptographic key as a 32-bit value). Others are not known and so you can call **BCryptGetProperty** with a NULL value for the pointer to the buffer and the function will return the required buffer size through the pointer used to indicate the buffer size. Changing a property is similar: you call the **BCryptSetProperty** [http://msdn.microsoft.com/en-us/library/aa375504(v=VS.85).aspx] function passing the provider handle, the name of the property, and a pointer to the buffer containing the new property value and use a parameter to indicate the size of this buffer.

Once you have set the properties of the algorithm you may call one of the CNG functions to perform the cryptographic action. If you wish to encrypt or decrypt data then you will need to provide a key. To do this you can either create a key with a call to the **BCryptGenerateSymmetricKey [http://msdn.microsoft.com/en-us/library/aa375453(v=VS.85).aspx]** function to create a symmetric (or secret) key or the **BCryptGenerateKeyPair** [http://msdn.microsoft.com/en-us/library/aa375451(v=VS.85).aspx] functions to create the asymmetric (or public-private) key pair. These functions return a **BCRYPT_KEY_HANDLE** handle and when you have finished with the handle you release the associated resources by calling the **BCryptDestroyKey** [http://msdn.microsoft.com/en-us/library/aa375404(v=VS.85).aspx] function. The key can then be passed, along with buffers for the input and output values to **BCryptEncrypt** [http://msdn.microsoft.com/en-us/library/aa375421(v=VS.85).aspx] to encrypt data or to **BCryptDecrypt** [http://msdn.microsoft.com/en-us/library/aa375421(v=VS.85).aspx] to decrypt data.

Creating a hash is a cryptographic operation and you can provide a key to be used by the algorithm, but this is optional. Before you can hash data you have to create a hash object by calling the **BCryptCreateHash** [http://msdn.microsoft.com/en-us/library/aa375383(v=VS.85).aspx] function, which returns a hash object handle that must be closed with a call to the **BCryptDestroyHash** [http://msdn.microsoft.com/en-us/library/aa375399(v=VS.85).aspx] function when you have finished performing the hash operation. You perform the hash by calling the **BCryptHashData** [http://msdn.microsoft.com/en-us/library/aa375468(v=VS.85).aspx] function passing the handle to the hash object and a pointer to a buffer with the data to hash (and a parameter indicating the size of the buffer). This function does not return the hashed data, in fact the data passed to the function is not modified, instead, the hash is maintained in memory by the hash object. The reason for this is you can call the **BCryptHashData** function more than once to hash the hashed data. To obtain the hashed value you call the **BCryptFinishHash** [http://msdn.microsoft.com/en-us/library/aa375443(v=VS.85).aspx] function passing the handle to the hash object and a user allocated buffer and its size, the function copies the hash from the hash object into the buffer. The size of the buffer is provided through the **BCRYPT_HASH_LENGTH** property.

Hashing Data in Hilo

Hilo provides a method called **FlickrUploader::CalculateMD5Hash** to generate an MD5 hash from a string passed as a parameter. The hash will be a binary value so the string returned from the **CalculateMD5Hash** method is the hex encoded value. Listing 6 shows the first half of this method, which creates the hash object. First the code calls the **BCryptOpenAlgorithmProvider** function to open the default MD5 provider. Next the method calls the **BCryptGetProperty** twice: the first time to get the size of the hash object and the second time to get the size of the buffer needed for the generated hash. These two buffers are created on the process heap by calling the **HeapAlloc** [http://msdn.microsoft.com/en-us/library/aa366597(VS.85).aspx] function but they could be allocated using any suitable memory allocator. Finally the **CalculateMD5Hash** method creates the hash object by calling the **BCryptCreateHash** function which initializes the buffer allocated for the hash object.

Listing 6 Creating a hash object

C++

```
std::wstring FlickrUploader::CalculateMD5Hash(const std::wstring& buffer)
{
   // Convert wstring to string
   std::string byteString(buffer.begin(), buffer.end());

   // Open an algorithm handle
   BCRYPT_ALG_HANDLE algorithm = nullptr;
   BCryptOpenAlgorithmProvider(&algorithm, BCRYPT_MD5_ALGORITHM, nullptr, 0);

   // Calculate the size of the buffer to hold the hash object
   unsigned long dataSize = 0;
   unsigned long hashObjectSize = 0;
   BCryptGetProperty(
      algorithm, BCRYPT_OBJECT_LENGTH, (unsigned char*)&hashObjectSize, sizeof(unsigned long),
&dataSize, 0);

   // Allocate the hash object on the heap
   unsigned char* hashObject = nullptr;
   hashObject = (unsigned char*) HeapAlloc(GetProcessHeap (), 0, hashObjectSize);

   // Calculate the length of the hash
   unsigned long  hashSize = 0;
   BCryptGetProperty(
      algorithm, BCRYPT_HASH_LENGTH, (unsigned char*)&hashSize, sizeof(unsigned long),
&dataSize, 0);

   // Allocate the hash buffer on the heap
   unsigned char* hash = nullptr;
   hash = (unsigned char*)HeapAlloc (GetProcessHeap(), 0, hashSize);

   // Create a hash
   BCRYPT_HASH_HANDLE cryptHash = nullptr;
   BCryptCreateHash(algorithm, &cryptHash, hashObject, hashObjectSize, nullptr, 0, 0);
```

The next part of the **CalculateMD5Hash** method is shown in Listing 7. This code calls the **BCryptHashData** function to hash the data passed as the second parameter. The hash object retains the actual hash which is then obtained by calling the **BCryptFinishHash** function passing the buffer previously allocated to hold the hash. Finally this binary data is converted to a hex string.

Listing 7 Hashing the data

C++

```
   // Hash data
   BCryptHashData(
      cryptHash, (unsigned char*)byteString.c_str(), static_cast<unsigned
long>(byteString.length()), 0);

   // Close the hash and get hash data
   BCryptFinishHash(cryptHash, hash, hashSize, 0);

   std::wstring resultString;
   // If no issues, then copy the bytes to the output string
   std::wostringstream hexString;
   for (unsigned short i = 0; i < hashSize; i++)
   {
      hexString << std::setfill(L'0') << std::setw(2) << std::hex << hash[i];
   }
   resultString =  hexString.str();
```

The last part of the **CalculateMD5Hash** method is shown in Listing 8. This shows the cleanup that is needed to

release the hash object and the cryptographic provider and to release the previously allocated memory.

Listing 8 Cleaning up the objects and buffers used by the method

C++

```
    // Cleanup
    BCryptCloseAlgorithmProvider(algorithm, 0);
    BCryptDestroyHash(cryptHash);
    HeapFree(GetProcessHeap(), 0, hashObject);
    HeapFree(GetProcessHeap(), 0, hash);

    return resultString;
}
```

Conclusion

In this chapter you have seen how the Hilo Browser uploads photos to the Flickr web site using the HTTP Services API to make a HTTP POST request, and how Hilo uses task dialogs to provide visual feedback about the upload operation to the user. You have also seen how the CNG functions can be used to create cryptographic hashes used by Flickr to verify the integrity of the parameters passed to it by a client. In the next chapter you will see how Hilo uses the Windows 7 Web Services API to call the Flickr Web Service to obtain an authentication session key and an access token.

Chapter 16: Using the Windows 7 Web Services API

The Hilo Browser application allows you to share your photos via Flickr by using the Share dialog. The previous chapter showed how the Share dialog uses the Windows HTTP Services API to upload the selected photos to Flickr using a multi-part HTTP POST request. Before the photo can be uploaded the Hilo Browser must first be authenticated with Flickr by obtaining a session token (called a *frob*), and then authorized to upload photos by obtaining an access token. To accomplish these two steps, Hilo Browser uses the Windows 7 Web Services Application Programming Interface (WWSAPI) to access Flickr using web services. In this chapter we will explore how the Hilo Browser uses this library.

Accessing Flickr through Web Services

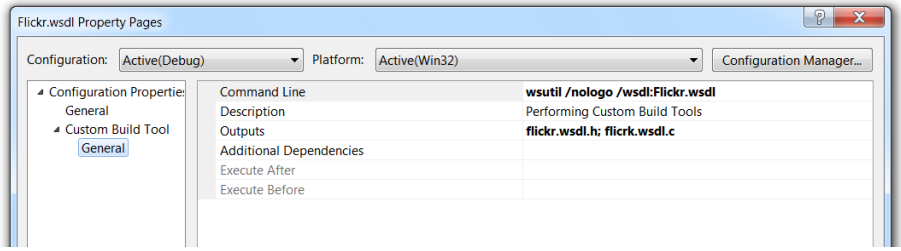
In the last chapter you saw how Hilo uploaded photos to the Flickr web server through a HTTP POST request. A key parameter of this request is the access token that indicates that the Hilo Flickr application is authorized to access the user's Flickr account. In addition, the Hilo Flickr account has to be authenticated with the Flickr web server to prove its identity, and from this Hilo obtains a session key called a frob. These two actions, obtaining the frob and the access token, are performed through web service calls.

Web services are a mechanism where client applications can communicate with web servers across the internet to make requests and receive data as a response. The key part of web services is the contract, a description of how the client makes a request and how the server responds, and the possible responses. Typically web services are implemented using XML messages over HTTP with the messages formatted as SOAP but there are other options. The contract between the client and server is immutable, if the server developers want to improve the service by adding more parameters or changing parameter types they have to provide a new web service with another contract, while still supporting the older web service.

This contract may be provided as a published specification and web service client developers must follow this specification to the letter. However, in some cases the contract may be provided as Web Services Description Language (WSDL) which is a standard XML description of a web service. WSDL can be used by code generation tools to generate proxy code. The proxy code does the work of constructing the packets sent to the service and decoding the packets returned from the service.

The Flickr web service is described on the Flickr API [http://www.flickr.com/services/api/] website. Flickr does not provide WSDL, instead it describes the request SOAP packet [http://www.flickr.com/services/api/request.soap.html] , the response SOAP packet [http://www.flickr.com/services/api/response.soap.html] and gives generic descriptions of the various API methods. From this it is possible to create a WSDL file, and the Hilo developers have done this to create the file **Flickr.wsdl** in the Browser project. The properties page for this file has a custom build step that uses a tool called **wsutil.exe** [http://msdn.microsoft.com/en-us/library/dd430644(VS.85).aspx] (Figure 1). This tool is part of the Windows 7 Software Development Kit (SDK) and so to use this tool you must have the SDK installed and the path to the SDK **bin** direct added to the project's executable path. The **wsutil** tool reads the WSDL file and generates C header and source files for accessing the web service.

Figure 1 Custom build steps for WSDL file



Authenticating a Flickr API Call

The Flickr API allows an application to access a Flickr account. There are two players in this transaction: the Flickr application and the user account, and both need to have a Flickr account. In Hilo this means that you have to create a Flickr account for Hilo and obtain a Flickr API key that identifies Hilo to Flickr. Hilo uploads photos to a user specified account and as part of this mechanism the user has to authorize Hilo and indicate to Flickr that Hilo can access the specified account. This involves several authentication and authorization steps.

When you apply for a Flickr API key (see Chapter 12 [http://msdn.microsoft.com/en-us/library/gg241211.aspx]) you will be given two strings: one is the API key and the other is a secret;. The API key identifies the user: the Hilo Flickr application. The secret is known only by you and Flickr and is used to create message digests (as explained in the last chapter) so that Flickr can verify that the parameters were not corrupted (either maliciously, or by accident) during transmission.

Before Hilo can upload a photo it has to logon to Flickr using its API key. To do this Hilo has to call the **flickr.auth.getFrob** [http://www.flickr.com/services/api/flickr.auth.getFrob.html] web service method with the API key as the parameter. This method returns a logon session key called a frob. The frob identifies the Hilo application just for the current session and is only valid for 60 minutes, but rather than holding onto a frob until it becomes invalid, Hilo takes the simpler path of obtaining a new frob every time the user wishes to upload a photo.

Hilo will upload photos to a Flickr account and so Flickr must be informed that all actions performed with the current session (identified through the frob) must occur on a specified account. The Hilo application does not know what Flickr account should be used when uploading photos, nor should it, Hilo should only upload using the current session and allow Flickr to determine where the data is stored. The way to do this is to create a Flickr logon URL containing the frob and the API key and launch this URL in a browser. The user is then able to log on to whatever account they wish and authorize the Hilo Browser to have access to the account in the current session. Once the user has authorized the Hilo Browser to access the account, you need to obtain an access token to upload photos. To do this you call the **flickr.auth.getToken** [http://www.flickr.com/services/api/flickr.auth.getToken.html] web service method passing the API key and the frob as parameters and receive back an access token.

Using the Windows 7 Web Services API

Hard coding the access to web services and the decoding of responses as illustrated by **FlickrUploader::UploadPhotos** and **FlickrUploader::GetPhotoId** methods described in the last chapter, is brittle. As the developer you have to be aware of every detail of the protocol so that the request is made in exactly the right format because a slight deviation in the format will result in the web service rejecting the request. There is also the question of what happens if the protocol changes. Once published, the interface of a web service should never change; however, during the development of a service the interface may change requiring changes to the client. This increases the chance of errors. In addition to the issue of writing the correctly formatted messages there is also the issue of managing the transport code.

The Windows 7 Web Services API provides a way to access web services without writing large amounts of code to create request packets, interpret response packets, or write transport code. The starting point of any web service client application is the WSDL specification for the service. The Hilo development team has created a WSDL file, **Flickr.wsdl**, using this information and you will find this file in the top level folder of the Browser project in Solution Explorer.

Examining the Web Services Description Language File

The WSDL document defines services as collections of network endpoints, or ports. Each port will have request messages in a specific format and will return the response and fault messages in a specific format. In defining the port you need to specify the format of the data used in a request and response, and you need to specify the actual endpoints that will be used. To build in flexibility, WSDL allows you to provide abstract entities that describe various aspects of the web service and use the abstract entities with real values and create one or more concrete entities. So a **portType** element is abstract and defines a web service method but not where the method is implemented; whereas a **port** element is a concrete entity because it uses the **portType** element and has the actual network endpoint where the web service method can be accessed.

The Flickr API defines a generic request format and response data format, and a generic data format for returning information about errors. These are described in the **Flickr.wsdl** file as the abstract **element** types: **FlickrRequest**, **FlickrResponse,** and **FlickrFault**. From these data types the **Flickr.wsdl** file declares the abstract **message** formats (the **FlickrRequestMessage**, **FlickrResponseMessage,** and **FlickrFaultMessage**

elements) that will be used to make the request and receive the responses.

In the **Flickr.wsdl** file two ports are described. The abstract **portType** elements are **FlickrFrobRequestPort** and **FlickrTokenRequestPort** which give details of the request, response, and fault messages used when accessing the ports (these definitions are the same, since the same request and response messages are used by both web service methods). In addition to the message formats, these **portType** elements give the name of the method to call (**flickr.auth.getFrob** and **flickr.auth.getToken**). These definitions are all abstract, they define the format of the request and response but they do not give details about the protocol, nor the actual network endpoint. The **binding**, **port** and **service** elements are concrete in that they contain information about the actual protocols and endpoints that will be used.

The **binding** elements, **FlickrFrobRequestPortBinding** and **FlickrTokenRequestPortBinding** reference the **portType** definition and provide information about the protocol that will be used. The binding is a concrete specification of the protocol and data format used for a particular **portType**. The **service** element contains the concrete definition of the collection of related endpoints, and each endpoint is given as a **port** element. In the **Flickr.wsdl** file the **flickr.auth.getFrob** and **flickr.auth.getToken** methods are declared as being part of two separate services, **FlickrFrobRequestPortService** and **FlickrTokenRequestPortService** respectively. Each service definition is a collection of **port** elements and each **port** element provides the **binding** element for a method and the endpoint where that web service method is implemented.

Generating the Proxy Code

The great advantage of using WSDL files is that they are compiled. You get two benefits here. First, the compiler will parse the WSDL file and issue errors if the WSDL code has invalid definitions. If you write the protocol code by hand you do not get this checking and so it is very easy to introduce logical errors that will be difficult to identify later. The second advantage is that the compiler will generate C code to access the web service and this reduces the amount of code that you have to write. Since the WSDL compiler, **wsutil**, generates C it means that you get high performance protocol binding code.

After you have created a WSDL file (or obtained one from a web service's website) and added it to the project as mentioned above, you should compile it to generate the C source file and header file for the proxy code. You then add the C file to the project so that you can use the proxy code. There is no need to alter the compiler options for this file since it is safe to compile it as a C++ file. However, be aware that since the code is inherently C code, the definitions created will be structures, and functions will depend upon opaque handles.

The WSDL compiler creates structures for the data types, messages, bindings, and service descriptions and creates instances of these structures populated with the data from the concrete definitions in the WSDL file. In addition the compiler generates functions to allow you to create the channel and the protocol bindings for each web service defined in the WSDL. These functions create a proxy to the web service, that is, a function that your code calls as if it is the web service method. The name of the proxy function is generated from the name of a **binding** element appended with **_CreateServiceProxy**. Since there are two **binding** elements in **Flickr.wsdl** there are two functions generated: **FlickrFrobRequestPortBinding_CreateServiceProxy** and **FlickrTokenRequestPortBinding_CreateServiceProxy**. Both of these functions return a handle that is a pointer to a **WS_SERVICE_PROXY** structure. This structure is allocated by the system so this handle must be released when you are finished making calls with a call to the **WsFreeServiceProxy** [http://msdn.microsoft.com/en-us/library/dd430534(VS.85).aspx] WSSAPI function.

web services may return rich error information through an error object. If you chose to receive this information you create an error object by calling the **WsCreateError** [http://msdn.microsoft.com/en-us/library/dd430497(v=VS.85).aspx] function that returns an opaque handle, a **WS_ERROR** pointer, that you can deallocate when you have finished by calling the **WsFreeError** [http://msdn.microsoft.com/en-us/library/dd430526(v=VS.85).aspx] function. You pass this handle to any method that can return error information and use the **WsGetErrorString** [http://msdn.microsoft.com/en-us/library/dd430540(v=VS.85).aspx] and **WsGetErrorProperty** [http://msdn.microsoft.com/en-us/library/dd430539(v=VS.85).aspx] functions to get information about the error. The error object can be reused by calling the **WsResetError** [http://msdn.microsoft.com/en-us/library/dd430614(v=VS.85).aspx] function, so you should only need to allocate the error object once and then deallocate it once you have finished making web service calls.

When the WWSAPI accesses a web service it will need to allocate various buffers. Rather than giving the responsibility to you to make the individual memory allocations (with the danger that you may forget to deallocate memory and cause memory leaks) the WSSAPI will do the allocations for you with a memory heap object. Your

only responsibility is to create the heap object before you make any web service calls by calling the **WsCreateHeap** [http://msdn.microsoft.com/en-us/library/dd430499(v=VS.85).aspx] function and to release the heap object with a call to the **WsFreeHeap** [http://msdn.microsoft.com/en-us/library/dd430527(v=VS.85).aspx] function when you have completed all web service calls.

The WSDL compiler creates functions to make the actual calls to the web service methods, with names derived from the binding elements and the method they bind to. For **Flickr.wsdl** the compiler creates two functions **FlickrFrobRequestPortBinding_flickr_auth_getFrob** and **FlickrTokenRequestPortBinding_flickr_auth_getToken**. These functions have parameters for the service proxy, the in parameters (used to make the request), pointers to buffers for the out parameters (for the response), handles for the heap object, and the error object that will be used.

Calling the Proxy Code

Hilo uses the WWSAPI in the **FlickrUploader** class, specifically the **GetToken** and **ObtainFrob** methods. To use WWSAPI you have to include the WebServices.h header file and link to the WebServices.lib library from the Windows 7 SDK, you also need to include the header file created by the **wsutil** tool from your WSDL file, and include the generated C file in the project.

Since there are two methods that use WWSAPI, the **FlickrUploader** class factors the code to initialize and clean up the proxy code in the private methods, **CreateWebProxy** and **CloseWebProxy**. **CreateWebProxy** creates a heap object and an error object and then creates and opens the proxy. To give more flexibility the **CreateWebProxy** method does not use the code generated by the **wsutil** tool to create the web proxy object, instead the WWSAPI functions are called directly. Although the **CreateWebProxy** method returns a handle to an error object and Hilo passes this to the web service proxy, the current version of Hilo only provides basic error handling and does not access the data in the error object.

The **CreateWebProxy** method creates the web service proxy by calling the **WsCreateServiceProxy** [http://msdn.microsoft.com/en-us/library/dd430507(VS.85).aspx] function indicating the type of channel to use, the relevant code is shown in Listing 1. The first parameter indicates that the proxy will be called to make requests and receive a response. The second parameter indicates that SOAP over HTTP will be used to access the service.

The **WsCreateServiceProxy** function can provide additional values through the sixth parameter which is an array of **WS_CHANNEL_PROPERTY** structures. Each of these has a property ID [http://msdn.microsoft.com/en-us/library/dd401786(v=VS.85).aspx] followed by a pointer to the property value and the size of the property. The **CreateWebProxy** method provides three properties, the first supplies the version of SOAP to use, the second indicates that no addressing-related headers are transmitted as part of the SOAP envelope. The final property specifies that all data is supplied as UTF8.

Listing 1 Creating a proxy

C++

```
WS_ENVELOPE_VERSION soapVersion = WS_ENVELOPE_VERSION_SOAP_1_2;
WS_ADDRESSING_VERSION addressingVersion = WS_ADDRESSING_VERSION_TRANSPORT;
WS_ENCODING encoding = WS_ENCODING_XML_UTF8;

WS_CHANNEL_PROPERTY channelProperties[3] =
{
    {
        WS_CHANNEL_PROPERTY_ENVELOPE_VERSION,
        &soapVersion, sizeof(WS_ENVELOPE_VERSION)
    },
    {
        WS_CHANNEL_PROPERTY_ADDRESSING_VERSION,
        &addressingVersion, sizeof(WS_ADDRESSING_VERSION)
    },
    {
        WS_CHANNEL_PROPERTY_ENCODING,
        &encoding, sizeof(WS_ENCODING)
    }
```

```
};

if(SUCCEEDED(hr))
{
   hr = WsCreateServiceProxy(
      WS_CHANNEL_TYPE_REQUEST, WS_HTTP_CHANNEL_BINDING,
      nullptr, nullptr, 0,
      channelProperties, ARRAYSIZE(channelProperties),
      proxy, // out parameter, returns the proxy
      *error);
}
```

A web service is implemented on an endpoint so you have to call the **WsOpenWebServiceProxy** [http://msdn.microsoft.com/en-us/library/dd430577(v=VS.85).aspx] function to provide this endpoint to the proxy. The code to do this in the **CreateWebProxy** method is shown in Listing 2.

Listing 2 Opening the proxy

C++

```
static const wchar_t* flickr_soap_endpoint_url = L"http://api.flickr.com/services/soap/";

WS_ENDPOINT_ADDRESS address =
{
   {
      static_cast<unsigned long>(wcslen(flickr_soap_endpoint_url)),
      const_cast<wchar_t*>(flickr_soap_endpoint_url)
   }
};

if(SUCCEEDED(hr))
{
   hr = WsOpenServiceProxy(*proxy, &address, nullptr, *error);
}
```

After you have created the web service proxy you can make calls to the web service by calling the **WsCall** [http://msdn.microsoft.com/en-us/library/dd430485(v=VS.85).aspx] function. The **WsCall** function has parameters that describe the web service operation to call and the parameters to be passed to the service. The function will return the results from calling the web service. The call to the **WsCall** method is provided by the functions generated by the **wsutil** tool from the WSDL file.

The **CreateWebProxy** method creates a heap and error object as well as the proxy object, and all of these are used to make calls to the web service. All of these objects allocate resources that must be released when you no longer need to use the proxy and this is done by calling the **CloseWebProxy** method, Listing 3.

Listing 3 Releasing resources

C++

```
void FlickrUploader::CloseWebProxy (WS_HEAP** heap, WS_SERVICE_PROXY** proxy,  WS_ERROR**
error)
{
    if (proxy != nullptr && *proxy != nullptr)
    {
        WsCloseServiceProxy(*proxy, nullptr, nullptr);
        WsFreeServiceProxy(*proxy);
    }

    if (heap != nullptr && *heap != nullptr)
    {
        WsFreeHeap(*heap);
    }
```

```
    if (error != nullptr && *error != nullptr)
    {
        WsFreeError(*error);
    }
}
```

Using the Windows 7 Web Services API in Hilo

The WWSAPI is used to make calls to the Flickr API web service to obtain the frob (the **ObtainFrob** method) and access token for the upload operation (the **GetToken** method). Listing 4 shows the basic code from the **GetToken** method to make the call: initialize the proxy; make the call; close the proxy.

Listing 4 The GetToken method

C++

```
std::wstring FlickrUploader::GetToken(const std::wstring& frob)
{
   std::wstring outputString;
   WS_ERROR* error = nullptr;
   WS_HEAP* heap = nullptr;
   WS_SERVICE_PROXY* proxy = nullptr;

   HRESULT hr = CreateWebProxy(&heap, &proxy, &error);

   // Call Web Service…

   CloseWebProxy(&heap, &proxy, &error);
   return outputString;
}
```

The code to call the web service is shown in Listing 5. The parameters of the method are the name of the method (**L" flickr.auth.getToken"**), the API key, the signature, and the frob. These parameters are passed through an instance of the **_FlickrRequest** structure generated by the **wsutil** tool. The signature is generated by concatenating the API key, frob, and method name and then calculating the MD5 hash.

Listing 5 Code to obtain a token

C++

```
if(SUCCEEDED(hr))
{
   _FlickrRequest request =
   {
      (wchar_t*)(flickr_auth_getToken_method_name),
      (wchar_t*) flickr_api_key, // api_key
      nullptr,                          // api_sig computed later
      (wchar_t*)frob.c_str()
   };

   std::wstring params = flickr_secret;
   params += L"api_key";
   params += request.api_key;
   params += L"frob";
   params += request.frob;
   params += L"method";
   params += request.method;

   std::wstring api_sig = CalculateMD5Hash(params);
   request.api_sig = const_cast<wchar_t*>(api_sig.c_str());
   wchar_t* token = nullptr;
   hr = FlickrTokenRequestPortBinding_flickr_auth_getToken(
```

```
        proxy, &request, &token, heap, nullptr, 0, nullptr, error);
    if (SUCCEEDED(hr))
    {
        bool errorFound = false;
        std::wstring value = GetXmlElementValueByName(token, L"token", &errorFound);
        if (!errorFound)
        {
            outputString = value;
        }
    }
}
```

The **GetToken** method then calls the **FlickrTokenRequestPortBinding_flickr_auth_getToken** function that was created by the **wsutil** tool. This function makes the call to the WSSAPI **WsCall** function and returns the results as a SOAP packet. Finally, the **GetToken** method calls the **GetXmlElementValueByName** method that uses XmlLite to obtain the value of the **token** element that is returned from the **GetToken** method.

Conclusion

In this chapter you saw how to use the Windows 7 Web Services API to make requests to the Flickr web services in order to authenticate and authorize the Hilo Browser so that it can upload the selected photos.

This chapter concludes this series of articles. In this series, we've described how the Hilo Browser and the Hilo Annotator were implemented using some of the powerful features and APIs provided by Windows 7. We've described how to design a touch-enabled user interface, and how to implement it using Direct2D, the Windows Animation Manager, and the Windows Ribbon. We've described how to integrate the application into the Windows Shell and how to use the Windows Imaging Component to manipulate images. And finally we've described how to share photos using HTTP and web services.

We hope you have enjoyed this series of articles, and we hope that they will help you to build rich, compelling Windows applications of your own.