WPF 4, Surface 2, and Kinect

Natural User Interfaces in .NET

Joshua Blake

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Natural User Interface version 8

For more information on this and other Manning titles go to
www.manning.com

TABLE OF CONTENTS

1

The natural user interface revolution

You are the person we've been waiting for. The people of the world are ready to change how they interact with computing devices. For years, futuristic computer interfaces in movies and academic research has teased the public while technology lagged. This gap has slowly been closing and now the technology of today has caught up with the interfaces of the future. Everything we have been imagining is now possible. All that is needed is someone to create software that takes advantage of the latest technology and realizes the future interface concepts. That someone is you.

You might wonder how you are going to fill this role. These futuristic interfaces seem impossible to create and the technology is fairly complex. I'm going to be your guide on these topics. I will teach you everything you need to know about creating multi-touch applications and natural user interfaces, which is what the industry is calling the next generation of human-computer interfaces.

You and I will start from the ground up, or if you prefer, from the fingertips out. I'll use simple examples to illustrate fundamental principles of natural user interfaces. After that, I will teach you how to use various APIs to implement the concepts you have learned. Finally, we will start building more complex and functional interfaces to give you a great starting point for developing your own natural user interface applications.

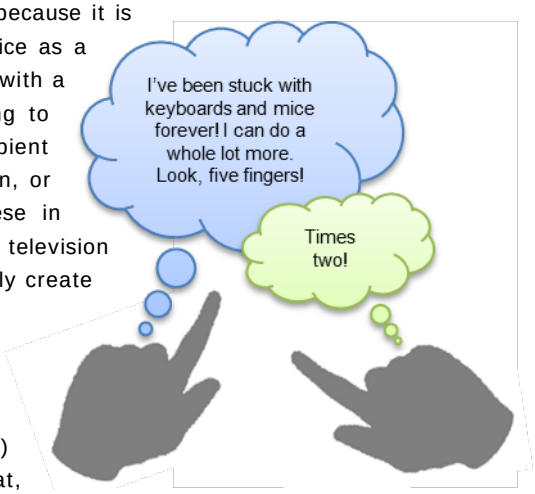
Before we get to the code, you will need to have a good understanding of what the natural user interface is. This chapter will give a good foundation for understanding the natural user interface. You will learn what it means for an interface to be natural, the role metaphor plays in human-computer interaction, what type of input technologies are useful in natural user interfaces, and finally we will wrap up this chapter with a review of the core principles of natural user interfaces. If you choose, I can prepare you to participate in the natural user interface revolution.

In the movie *The Matrix*, when Neo first met Morpheus, Morpheus gave him a choice. On one hand, Neo could choose stay in the world he was comfortable with, the world he had known his whole life. On the other hand, Neo could choose to open his eyes and discover a new way to think about his world, and ultimately take part in a revolution. I offer you the same choice. If you want to go back to your familiar yet aging computing experience with the keyboards and mice, windows and menus, then stop reading now. Instead, if you want to expand your mind and learn how to be an active part of the natural user interface revolution, then ask me the question that brought you here and you will start your journey.

1.1 What is the natural user interface?

This is the question we all start with, so I'm glad you decided to continue. Before I give you a definition, allow me to describe the role and context of the natural user interface. The natural user interface, or NUI (pronounced "new-ee"), is the next generation of interfaces. We can interact with natural user interfaces using many different input modalities, including multi-touch, motion tracking, voice, and stylus. It is true that NUI increases our input options, but NUI is about more than just the input. NUI is a new way of thinking about how we interact with computing devices.

The hand images will have different poses later. For now I'm just reusing the one pointing pose for planning.



I say computing devices here instead of computers because it is not a given that we will think of every interactive device as a computer in the future. We might use a NUI to interact with a flexible e-ink display that wirelessly off-loads computing to the cloud, or we might use a NUI to interact with ambient devices that are embedded in our clothes, our television, or our house. You have probably seen devices like these in movies such as *Minority Report* and *The Island* and or television shows such as *CSI*, but now you are preparing to actually create these interfaces.

There are several different ways to define the natural user interface. The easiest way to understand the natural user interface is to compare it to other type of interfaces such as the graphical user interface (GUI) and the command line interface (CLI). In order to do that, let's reveal the definition of NUI that I like to use.

DEFINITION: NATURAL USER INTERFACE

A natural user interface is a user interface designed to reuse existing skills for interacting directly with content.

There are three important things that this definition tells us about natural user interfaces.

NUIs ARE DESIGNED

First, this definition tells us that natural user interfaces are designed, which means they require forethought and specific planning efforts in advance. Special care is required to make sure NUI interactions are appropriate for the user, the content, and the context. Nothing about NUIs should be thrown together or assembled haphazardly. We should acknowledge the role that designers have to play in creating NUI style interactions and make sure that the design process is given just as much priority as development.

NUIs REUSE EXISTING SKILLS

Second, the phrase "reuse existing skills" helps us focus on how to create interfaces that are natural. Your users are experts in many skills that they have gained just because they are human. They have been practicing for years skills for human-human communication, both verbal and non-verbal, and human-environmental interaction. Computing power and input technology has progressed to a point where we can take advantage of these existing non-computing skills. NUIs do this by letting users interact with computers using intuitive actions such as touching, gesturing, and talking, and presenting interfaces that users can understand primarily through metaphors that draw from real-world experiences.

This is in contrast to GUI, which uses artificial interface elements such as windows, menus, and icons for output and pointing device such as a mouse for input, or the CLI, which is described as having text output and text input using a keyboard.

At first glance, the primary difference between these definitions is the input modality -- keyboard versus mouse versus touch. There is another subtle yet important difference: CLI and GUI are defined explicitly in terms of the input device, while NUI is defined in terms of the interaction style. Any type of interface technology can be used with NUI as long as the style of interaction focuses on reusing existing skills.

NUIs HAVE DIRECT INTERACTION WITH CONTENT

Finally, think again about GUI, which by definition uses windows, menus, and icons as the primary interface elements. In contrast, the phrase "interacting directly with content" tells us that the focus of the

interactions is on the content and directly interacting with it. This doesn't mean that the interface cannot have controls such as buttons or checkboxes when necessary. It only means that the controls should be secondary to the content, and direct manipulation of the content should be the primary interaction method. We will discuss different types of directness in section 1.3.4, but direct applies to both touch interfaces as well as motion tracking and other NUI technologies.

Now that you have a basic understanding of what the natural user interface is, you might be wondering about the fate of graphical user interfaces.

1.1.1 Will NUI replace GUI?

There is often confusion about the future of GUI when we talk about NUIs. To help answer this question, let me pose a different question:

RHETORICAL QUESTION

Did the graphical user interface replace the command line interface?

If you answered yes, then you are correct. In its prime, the command line interface and textual interfaces were used for many different purposes ranging from word processing to accessing network resources (such as bulletin board systems) to system administration. Today all of those tasks are handled by GUI equivalents. GUI effectively took over the role of the CLI.

On the other hand, if you answered no, you would still be correct. The CLI is still used for specialized tasks where it is best at, primarily certain system administration and programming tasks as well as tasks that require scripting. The CLI is still around and today you can still access it in Windows with only a few keystrokes or clicks.

The command line interface was once for general purpose tasks, but is now limited to specialized tasks that are most effective with a CLI. The GUI took over the general purpose role, and at the same time the new capabilities allowed computing applications to grow far beyond what the CLI was capable of. The total scope of the computing world is orders of magnitude larger today with GUI than when CLI was king.

Now the same pattern is occurring with NUIs. The natural user interface will take over the general purpose role from graphical user interfaces, but GUIs will still be around for when the GUI is the most effective way to accomplish a specialized task. These tasks will likely be things that require precise pointing capability such as graphic design as well as data entry-heavy tasks. I suspect that even those applications will be influenced by new interaction patterns that become popular with NUIs, so some future applications may look like hybrid GUI/NUI apps.

1.1.2 Why bother switching from GUI to NUI?

The natural user interface revolution is inevitable. Here again, history rhymes. The world migrated from CLI application to GUI applications because GUI was more capable, easier to learn, and easier to use in everyday tasks. From a business perspective, if you had a text-based application and your competitor created a new GUI application that everyone wanted, the market forces would demand you do the same. The same is happening again right now, except today GUI is the stale technology and NUI is the more capable, easier to learn, and easier to use technology.

History Rhymes: Interface transitions

While researching this book, I read several interesting research papers describing the pros and cons of various interface styles. Consider this quote from a paper describing the benefits of direct manipulation interfaces, which as we discussed above is an important aspect of natural user interfaces.

The quote (numbers 1-6) below should be inside the sidebar

1. Novices can learn basic functionality quickly, usually through a demonstration by a more experienced user.

2. Experts can work extremely rapidly to carry out a wide range of tasks, even defining new functions and features.

3. Knowledgeable intermittent users can retain operational concepts.

4. Error messages are rarely needed.

5. Users can see immediately if their actions are furthering their goals, and if not, they can simply change the direction of their activity.

6. Users have reduced anxiety because the system is comprehensible and because actions are so easily reversible.

This is a great list of attributes of natural user interfaces, except for one thing: the quoted paper was written in 1982 by Ben Shneiderman and he was describing the up-and-coming graphical user interface as a direct manipulation interface.

When considered in this perspective, GUI is more direct than the CLI, but in turn NUI is more direct than GUI. Today we don't think of GUI as very direct, partially because we are used to GUI style interactions and partially because of suboptimal interaction patterns that GUI application designers have adopted and repeated have not lived up to the promise of direct manipulation. For example, the idea that GUIs rarely need error messages, as implied by the quote above, is laughable.

NUI has several advantages over GUI for general purpose tasks. New input technologies make NUI more flexible and capable than GUIs, which are limited to keyboard and mouse. The focus on natural behaviors makes NUIs easier to learn than GUI, and everyday tasks are simpler to accomplish.

GUIs and mouse-driven interfaces will still have a role in the future, but that role will be more limited and specialized than today. If you are deciding between creating a GUI-style application or a NUI-style application, here are some simple yes-or-no questions you should ask:

- Will the application be used solely on a traditional computer with mouse and keyboard?
- Does the application need to be designed around mouse input for any reason besides legacy?
- Will the application be used only by well-trained users?
- Is the legacy of windows, menus, and icons more important than making the application easy to learn?
- Is task efficiency and speed of user transactions more important than the usability and user experience?

If the answer to any of these is no, then the application would benefit from using a natural user interface. In some cases, the mouse or similar traditional pointing device may be necessary for precise pointing tasks, but the overall application can still be designed as a natural user interface. Remember, NUI is not about the input. NUI is about the interaction style. It would be valid to design a NUI that used keyboard and mouse as long as the interactions are natural.

1.2 What does natural really mean?

I keep saying "natural" but haven't yet talked about what natural really means. In order to understand the natural user interface, you have to know what natural means. Many people use for "intuitive" interchangeably with "natural." Intuitive is an accurate description but is not any more revealing than "natural" about the nature of NUIs.

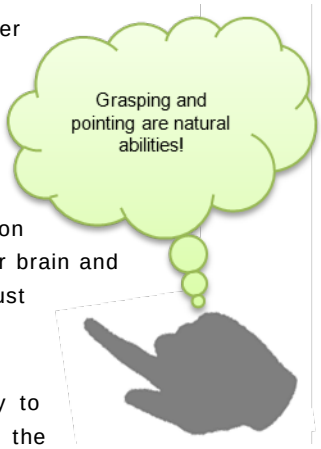
To understand what "natural" means, let's turn to Bill Buxton, one of the world's leading experts in multi-touch technologies and natural user interfaces.

An interface is natural if it "exploits skills that we have acquired through a lifetime of living in the world."

Bill Buxton, January 6, 2010, http://channel9.msdn.com/posts/LarryLarsen/CES-2010-NUI-with-Bill-Buxton/

This description is interesting for two reasons. First, it links the concept of natural with the idea of reusing existing skills. Second, it makes it explicit that these skills are not just the innate abilities we are born with. Natural means using innate abilities plus learned skills we have developed through interacting with our own natural environments in everyday life.

1.2.1 Innate abilities and learned skills explained



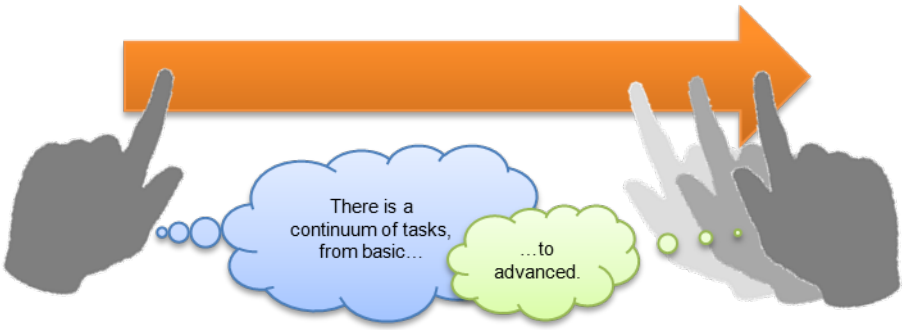
We are all born with certain abilities, and as we grow up certain other abilities mature on their own. Some examples of these abilities are eating, walking, and talking. We also have some low-level abilities hard-wired into our brains used for the basic operations of our bodies, as well as perception of our environment. A few examples of these low-level abilities are the ability to detect changes in our field of vision, perceive differences in textures and depth cues, and filter a noisy room to focus on one voice. You could say that innate abilities are like device drivers for our brain and body. The common thread is that we gain innate abilities automatically just by being human.

THE ABILITY TO LEARN

Humans also have one, very important ability: we have an innate ability to learn, which lets us add skills to our natural abilities. Learning is core to the human experience. We need to learn new skills in order to cope and adapt to our environment. Learned skills are different than innate abilities because we must choose to learn a skill, whereas abilities mature automatically. Once we learn a skill, it becomes natural and easy to repeat, as long as we maintain the skill.

Skills and abilities are used for accomplishing tasks. A task is a unit of work that requires user action and a specific result. Tasks may be composed of sub-tasks. One example of a user interface task is sending an email. In order to send an email, you must perform a set of sub-tasks such as creating a new message, setting the "to" and "subject" fields, typing the body of the email, then pressing send. We use specific skills to accomplish each sub-task as we progress towards the overall goal of sending the email. Tasks and skills go hand-in-hand because a task is something that needs to be done to achieve a result whereas a skill is our ability to do the task.

Tasks vary in difficulty from basic to advanced. Some skills only enable you to perform basic tasks, while other skills enable more advanced tasks.



We learn skills by building upon what we already know how to do. Humans start out with innate abilities and use them to learn skills to perform basic tasks. We progressively learn how to accomplish advanced tasks by building upon the existing skills. There are two categories of skills: simple and composite. Simple skills build directly upon innate abilities, and composite skills build upon other simple or composite skills.

In general, composite skills are used for more advanced tasks than simple skills, but there is a lot of overlap. Simple and composite skills each have different attributes and uses, so let's review them.

1.2.2 What are simple skills?

Simple skills are learned skills that only depend upon innate abilities. This limits the complexity of these skills, which also means simple skills are easy to learn, have a low cognitive load, and can be reused and adapted for many tasks without much effort. The learning process for simple skills is typically very quick and requires little or no practice to achieve an adequate level of competence. Many times learning can be achieved by simply observing someone else demonstrate the skill once or twice.

I have three young daughters, the oldest is four years old, and my wife and I like to play little games with them that help them learn. One of the games we like to play is sniffing flowers, as shown in figure 1.1. When we see flowers I'll ask them to sniff, and they will lean over and scrunch up their noses and smell the flower. Sometimes we will sniff other things, too, like food, toys, or even each other. That leads to another game where I'll sniff their bare feet, claim they are stinky (even when they are not), and use that as an excuse to tickle them, and then they will do the same to us.

Figure 1.1 Sniffing flowers is a simple skill because it only depends upon innate abilities

Sniffing is a simple skill. If you think about the requirements for the skill of sniffing, you will see that it only requires these abilities: conscious control of breathing, gross control of body motion, and perhaps the ability to wiggle your nose if you are an adorable toddler trying to be cute. Accordingly, sniffing exhibits all the attributes of a simple skill:

- *Sniffing is easy to learn.*

  My daughters learned the sniffing game by watching my wife and I demonstrate a few times.

- *Sniffing has low cognitive load.*

  It is easily performed by a toddler while doing other tasks such as interacting with her parents.

- *Sniffing can easily be adapted for new tasks.*

  Applying the already learned skill of sniffing to the stinky feet game was completely natural and required no prompting.

Tapping is a simple skill and is a natural human behavior that can be easily used in user interfaces. In order to master the skill of tapping, you only need to have the innate ability of fine eye-hand coordination. Tapping has all of the attributes of a simple skill: you can learn it easily through observation, you can tap while doing other things at the same time, and tapping is easily reused for different types of tasks such as calling attention to an object, keeping time with music, or pushing buttons on a television remote control. In a user interface tapping can be used for many tasks such as button activation or item selection.

You may ask why I used tapping as an example here rather than clicking with a mouse, when in most user interfaces you can do the same tasks with a single tap and a click. Even though the end result is the same, the action is not. Clicking with a mouse is a composite skill. Let's discuss what composite skills are and then revisit the mouse to discuss why it is composite.

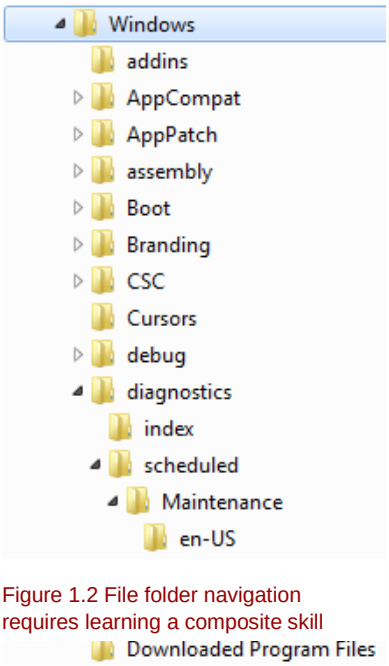


Figure 1.2 File folder navigation requires learning a composite skill

1.2.3 What are composite skills?

Composite skills are learned skills that depend upon other composites or skills, which means they can enable you to perform complex, advanced tasks. It also means that relative to simple skills, composite skills take more effort to learn, have a higher cognitive load, and are specialized for a few tasks or a single task with limited reuse or adaptability. Composite skills typically require learning and applying specific concepts that ties together the requisite skills. In order to achieve an adequate level of competence with composite skills, both conscious effort and practice are typically required.

One example of a composite skill common in GUI applications is the navigating file folders such as the folder tree shown in figure 1.2. This skill is a necessary part of the GUI experience if you want to do anything with opening or saving files. Although files and folders have a real-world equivalent, the actual implementation is so far from the real world it requires special effort to learn, and specific explanations of the file-folder metaphor. For example, in the real world, you do not often nest folders inside of other folders several layers deep. You might have folders in a filing cabinet, but applying that concept to the folder tree interface requires explanation.

Navigating folders or a folder tree requires a significant mental effort, even for advanced users. Rather than becoming an automatic behavior, expanding and collapsing and navigating folders remains something that requires significant attention each time.

Finally, folder navigation is not a pattern you will see reused much in other situations because it is so specialized. Even if an application uses a similar metaphor for organizing content, menu options, or configuration settings, users must relearn how to use the control for the specific application and the specific purpose.

Going back to our previous example, clicking with a mouse is a composite skill because it depends upon the skills of holding and moving a mouse and acquiring a target with a mouse pointer. Using those two skills together requires a conceptual understanding of the pointing device metaphor.

For you and me, using a mouse is easy. We don't even think about the mouse when we are using it, so you may wonder how this can be a composite skill. Here's why:

- *More effort to learn*

  We must invest a lot of practice time with the mouse before we can use it quickly and accurately. If you ever watch young children using a mouse, watch how much they overshoot and overcorrect before settling on their target.

- *Higher cognitive load*

  Mouse skills fall towards the basic side of the skill continuum, but the mouse still demands a measurable amount of attention.

- *Specialized with limited reuse*

  While tapping can be used for many different real-world and computing tasks, the master mouser skills you spent so much time developing have no other applications besides cursor-based interfaces.

I want to follow up on the subject of cognitive load. Consider this: while driving on a highway, changing the radio station is a fairly safe task using physical buttons and dials, but would be dangerously distracting if it required a mouse, cursor, and virtual buttons. Why is this?

1.2.4 Using skills increases cognitive load

We use skills all the time (in fact, you are exercising several skills right now while reading this), and using skills increases our cognitive load. In the example above, using a mouse-based interface to change a radio station would require more focus and higher cognitive load than reaching over and using physical controls.

What is cognitive load?

Cognitive load is the measure of the working memory used while performing a task. The concept reflects the fact that our fixed working memory capacity limits how many things we can do at the same time. Cognitive load theory was developed by John Sweller when he studied how learners perform problem-solving exercises.

Sweller suggested that certain types of learning exercises used up a significant portion of our working memory and since working memory is limited, the working memory is not available for the actual learning task. This means that learning is not as efficient when the exercises use too much memory.

There are three different types of cognitive load:

This list should be in the sidebar.

1. *Intrinsic cognitive load*—The inherent difficulty of the subject matter. Calculus is more difficult than algebra, for example.
2. *Extraneous cognitive load*—The load created by the design of the learning activities. Describing a visual concept verbally involves more cognitive load than demonstrating it visually.
3. *Germane cognitive load*—The load involved in processing and understanding the subject matter. Learning activities can be designed help users understand new material by preferring germane load.

Cognitive load theory states that extraneous load should be minimized to leave plenty of working memory for germane load, which is how people learn. The inherent difficulty of a subject cannot be changed, but the current intrinsic load can be managed by splitting complex subjects into sub-areas.

Cognitive load theory was originally applied to formal instruction but can be applied to different fields. I am using it to evaluate user interfaces because learning activities and human-computer interaction have many parallel issues. Table 1.1 shows the three types of cognitive load in the context of human-computer interaction and interface design.

Table 1.1 Cognitive load measures how much working memory is being used. If we consider the three types of cognitive load in terms of human-computer interaction it will help use create easier to use interfaces.

Cognitive load type	*HCI description*	*Example*
Intrinsic	The inherent difficulty of the task.	Interaction design cannot change the difficulty, but difficult tasks can be split into sub-tasks.
Extraneous	The load created by the skills used in the interaction.	A poorly designed interaction can make the user think more than necessary while a well-designed interaction can seem completely natural.
Germane	The load involved in learning the interface.	Well-designed interfaces focus on progressively teaching the user how to use it.

Managing cognitive load is like managing cholesterol levels. There is bad cholesterol you want to minimize (extraneous load), good cholesterol you want to maximize (germane load), and in general you want to keep the total cholesterol at a reasonable level (intrinsic load.)

In this context, we can see why we should use simple skills rather than composite skills when designing an interface. Composite skills depend upon several other skills and require conceptual thought, which generates extraneous load and reduces the working memory available for the intrinsic load of the actual task and the germane load for learning how to use the rest of the application. This results in users who are constantly switching between making high-level decisions about their tasks and figuring out the interface.

In contrast, simple skills have a low extraneous load because they are built upon the largely automatic innate abilities. This leaves most of the working memory available for the germane load of learning more advanced aspects of the interface as well as the intrinsic load generated by the actual task. This means the users can focus on high-level thoughts and decisions and on getting better at what they do rather than on using the interface itself.

The problem with cognitive load

You might wonder why I am stressing the importance of cognitive load in interface design. Even if a composite skill takes up more cognitive load than a simple skill, as long as it fits within my working memory it doesn't matter, right?

Wrong! Software developers have a tendency to think that if it works for me then it must work for everyone. This is true in application development, when a developer creates a fancy program that works great on his super powerful development machine with lots of RAM but is really slow on a normal machine, as well as interface design, when a developer creates an interface that makes sense to him but not the rest of the world. Cognitive load takes up working memory, which is like RAM in this way: not everyone has the same amount of working memory, and not everyone wants to devote their entire working memory to operating an interface.

There are many potential users who are able to use computers but have a limited amount of working memory. This includes young children with still developing brains, older people with deteriorating capacities, and people who have a mental handicap of some type. It also includes people who need to use an interface but need to remain aware of a hostile environment, such as a factory floor, a battle field, or even the road while driving a car. Unless you want to artificially limit who can use your interface effectively, then you should strive to minimize cognitive load.

Minimizing cognitive load also provides an advantage for those who are fortunate to have full use of their brain's potential. The less working memory it takes to operate your interface, the more working memory is available for higher-order brain functions. This would be a benefit for people trying to use your application to make complex decisions. A perfect example would be a financial analyst looking at

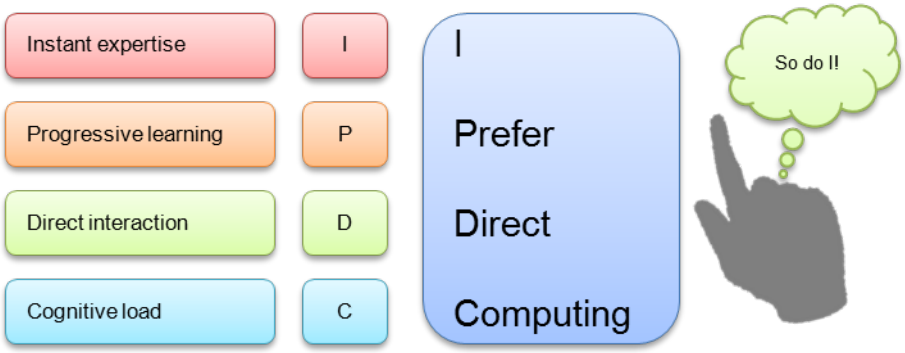
stock market trade data. The analyst is trying to find trends and perform complex, high-level decisions. The more room your interface leaves to those thought processes, the more effective he will be.

We have covered a lot of different concepts now about the nature of natural. Let's circle back and draw some conclusions about how these ideas apply to natural user interfaces.

1.3 Natural interaction guidelines

Based upon what we have discussed about innate abilities, simple and composite skills, and the types of cognitive load, we can derive four simple guidelines to follow that will help you design interactions that are natural. Those of you familiar with the Microsoft Surface User Experience Guidelines will see some cross-over with these guidelines. I have tried to create concise guidelines derived from an understanding of human cognition and natural interaction that can be applied to any type of natural user interface, regardless of the input modality. We will be referring to these guidelines throughout this book and once you get a feel for them, they will help you ask the important NUI design questions and hopefully lead you in the right direction towards an answer.

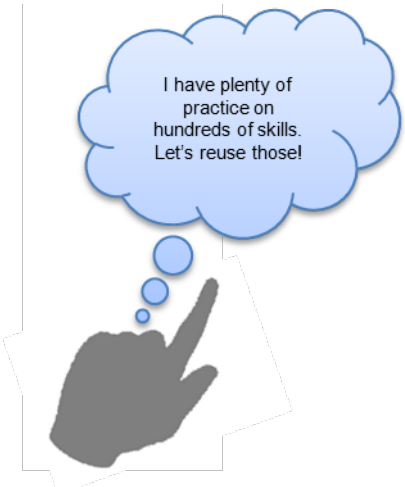
The names of the four guidelines are Instant expertise, Progressive learning, Direct interaction, and Cognitive load. In order to help you remember these guidelines, you could use this mnemonic device: "I Prefer Direct Computing."



"I Prefer Direct Computing" is perhaps a good motto for NUI enthusiasts and gives you a jump-start on remembering the natural interaction guidelines. Let's discuss what each of these guidelines are.

1.3.1 Instant expertise

This guideline states that you should design interactions that reuse existing skills. By doing this, you can take advantage of the investment your users made in their existing skills and create instant experts.

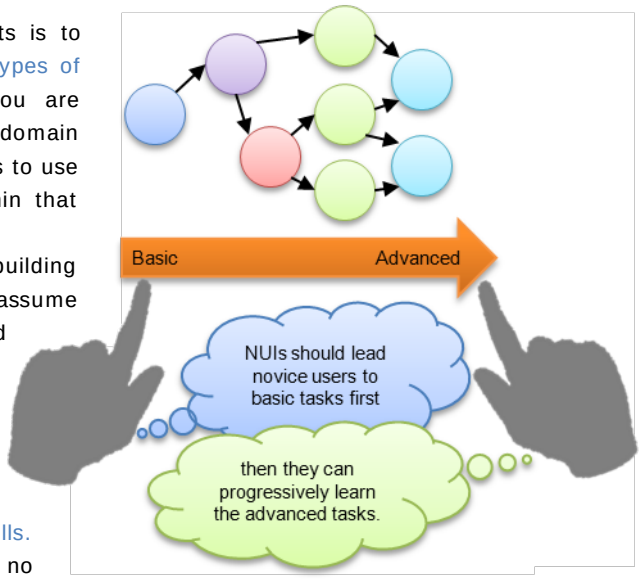


In both the real-world and user interfaces, the hardest part of using a skill is the learning process. Once a skill is learned it is significantly easier to exercise it. Your users already have hundreds or thousands of skills before they use your application, many of which they have been using since early childhood. If you can take advantage of an existing skill, then your users will not have to learn something new, they only have to apply the skill to the new situation. It is much easier for users to reuse existing skills than learn new skills. If you design your application interactions to reuse existing skills, your users can get up to speed very quickly with little effort and they will love you for it, even if they don't know why.

There are two ways to create instant experts by using skills your users already have: reuse domain-specific skills and reuse common human skills.

REUSING DOMAIN-SPECIFIC SKILLS

The first approach to creating instant experts is to reuse domain skills that are specific to the types of users who will use your application. If you are creating an application for people with specific domain knowledge, then you can figure out some skills to use based upon the activities your users do within that domain.



For example, if your application is a building design tool for civil engineers, then you might assume they already know how to read blueprints and how to use certain real-world tools. You could use this information to design your application, but there are a few problems with this approach.

The first problem is that not all users in your target group will have the same exact skills. An engineer fresh out of college with no experience may not have the skills to apply to your application, and different engineers may have different skill sets based upon the tools they use. The second problem is that most domain-specific skills are composite skills, which as we discussed are hard to apply to new situations. Reusing specialized composite skills would be difficult and could result in overly literal interface metaphors.

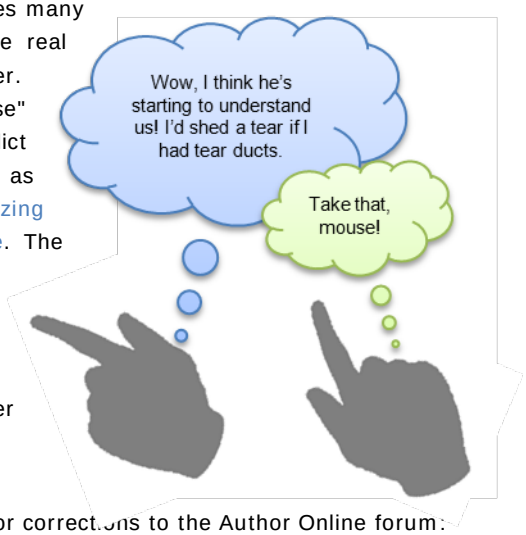
REUSING COMMON HUMAN SKILLS

The second approach to creating instant experts, and perhaps a better approach for most scenarios, is to assume that your users are human. You are designing interfaces for humans, right? Ok, now that we know humans will be using your application, we can reuse simple skills that your users have developed by taking part in the human experience. In chapters 2 and 3 we will talk about how the objects, containers, gestures, and manipulations concepts can help you create NUIs that use skills common to the human experience.

The instant expertise guideline tells us to reuse existing skills, but the next guideline provides contrasting advice that helps to balance designs.

1.3.2 Cognitive load

We discussed minimizing cognitive load in section 1.2.4 and it is important enough to be included as a guideline. The cognitive load guideline states that you should design the most common interactions to use innate abilities and simple skills. This will have two benefits. First, the majority of the interface will have a low cognitive load and be very easy to use. Second, the interface will be very quick to learn, even if some or all of the skills are completely new. If the interface uses many simple skills based on our natural interactions with the real world, the frequency of interaction can also be much higher.



You may wonder how to balance the "instant expertise" guideline with this guideline. These guidelines could conflict if the users already have a useful composite skill, such as using the mouse. In general you should focus on minimizing the cognitive load for as many interactions as possible. The best case is reusing existing simple skills. For example, putting ergonomics aside for the moment, a touch-based interface that uses existing abilities and simple skills is preferable to a mouse-based interface that uses existing composite skills. Using the mouse has a higher cognitive load than using your fingers.

When reusing simple skills is not possible, you should give priority to teaching simple skills rather than reusing composite skills. In the long run, teaching and using new simple skills requires less effort and is more natural than reusing composite skills. As an example, using a simple touch gesture to trigger an action would be preferable to navigating a menu with a mouse, even when you consider the small amount of effort necessary for learning the gesture.

The instant expertise and cognitive load guidelines help us design individual tasks, but the next guideline discusses the sequence of tasks within an application.

1.3.3 Progressive learning

This guideline states that you should provide a smooth learning path from basic tasks to advanced tasks. In real life, we start out with a few abilities then progressively learn basic skills and eventually master more advanced skills. This progressive learning curve lets us achieve small victories and use what we know while we continue to learn. If life required us to be experts when we were born, no one would last for long.

Natural user interfaces should enable the user to progressively learn and advance from novice to expert. At the same time, the interface should not get in the way of expert users doing advanced tasks. The preferred way to handle advanced tasks is to break them down into subtasks that use simple skills. In some cases this may not be possible and a complex but important task may require a composite skill. That is okay, as long as these complex tasks are limited in number and not part of the core interface used by beginning users. This will allow users to start using the application while working up to the advanced tasks.

THE PATH OF LEARNING

A key component of this guideline is ensuring that novice users are required to learn how to perform the basic tasks of the interface before being bombarded with more complex tasks. This implies there is a path through the interface that leads users by basic tasks on the way to advanced tasks. This is very similar to game design where the first level or two are designed to get the player acquainted with the controls and basic game behavior. As the levels progress, the player is progressively challenged with harder tasks and occasionally learns a new skill or ability.

It probably is not appropriate to design regular applications around levels, but what you can do is limit the number of tasks the user can do at any one time. Many complex GUI applications have so many toolbars and cascading menus it is difficult for new users to know what to do. Reducing the number of options will limit the number of interface elements necessary and result in fewer paths the user can take, but the user can make easier decisions about what to do next.

So far the guidelines have been about skills and tasks. The final guideline gives us advice on the design of the interaction quality.

1.3.4 Direct interaction

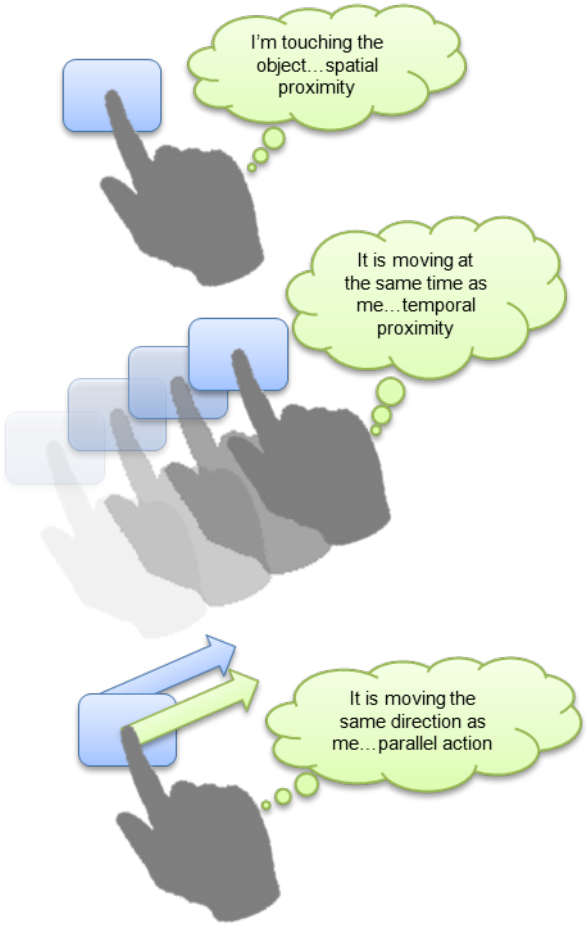
This guideline states that you should design the interface to use interactions that are direct, high-frequency, and appropriate to the context. Our interaction with the real world has these qualities, so this will result in interfaces that feel more fluid and natural, and will also allow users to access many features without overwhelming the user by presenting them all at once.

When you design direct interactions, you also end up getting high-frequency and contextual interactions. Let's talk about the different types of directness, and then what high-frequency interactions and contextual interactions mean.

TYPES OF DIRECTNESS

Drawing from our definition of natural user interfaces, it is important to enable the user to interact directly with content. There are three ways that an interaction can be direct:

- *Spatial proximity*—The user's physical action is physically close to the element being acted upon
- *Temporal proximity*—The interface reacts at the same time as the user action
- *Parallel action*—There is a mapping between at least one degree-of-freedom of the user action and at least one degree-of-freedom of the interface reaction



Different degrees of directness are possible depending upon the specific input modality. In many cases, using direct interaction eliminate the need for certain interface elements. To use a common example, consider an interface on a multi-touch screen that allows the user to pan, rotate, and scale images using direct touch manipulation without the need for dialog boxes or resize handles. This illustrates all three aspects of directness: the user touches the visual directly, the interface reacts immediately, and the horizontal and vertical motion of the fingers is mapped directly to horizontal and vertical movement of the visuals. The rotate and scale manipulations are mapped from the orientation of and distance between fingers, respectively.

Directness can apply to other input modalities as well. The same interface as above could be used with a device that tracks the motion of fingers in three-dimensional space. This would still be considered direct interaction, even though the fingers are not touching the visuals, because there is still temporal proximity and parallel action. If the user had an augmented reality display with this motion tracking, then spatial proximity could be restored.

A voice-based natural user interface could be temporally direct if it allowed the user to speak to interrupt the computer as it reads a list of items. It could also have parallel action if the user could speak in spatial terms to arrange visual elements on a display, or if the rate of speech or the pitch of the user's voice was mapped to an interface response. Even more interesting scenarios can be enabled by combining voice and touch.

Direct interaction with content is a natural extension of our rich sensory experience embodied within the real world and our ability to touch and manipulate real life objects. Interactions that are direct also tend to be faster than interacting indirectly with content through a mouse and interface elements.

If you design for direct interaction then each interaction will be much smaller, which means the interactions can be high-frequency.

HIGH-FREQUENCY INTERACTION

Interfaces that allow many quick interactions provide a more engaging and realistic experience. Each individual interaction may have only a small effect, but the bite-sized (or finger-sized, perhaps)

interactions are much easier and quicker to perform. This allows the user to perform many more interactions and get much more feedback in the process than a GUI-style interface would allow.

In real-world human-environmental interaction, our bodies are constantly in motion and we get back a constant stream of information from our senses. The feedback from all of these interactions is important. In a GUI you often have fewer, more significant actions but you only get large chunks of feedback at the end. It takes a lot of mental effort to process large chunks of feedback, especially if it is textual. In contrast, smaller chunks of feedback from small interactions can be processed quickly and the user can make subtle adjustments as necessary.

You can take advantage of this channel of communication to give your users extra contextual clues about the content they are interacting with. This allows a much richer interaction than binary success/failure feedback common in GUIs. You can help your user understand why they can or cannot do something. Using a real-life example, if you try to open a door but it is stuck on something, the door may have a little give that can help you track down why it is stuck. In the same way, when you're interacting with an interface element and you exceed a boundary, you have an opportunity to add feedback that lets the user know about the boundary but also that the interaction is still active.

All that high-frequency feedback would be overwhelming if it did not occur in a specific context.

CONTEXTUAL INTERACTIONS

The context of an interaction includes what action the user is performing, the proximity of the action to certain visuals, and the previous interactions before the current one. In order to create contextual interactions and have the feedback interpreted properly, it is important to reduce the number of possible tasks to only those that are appropriate at the time. Fewer tasks also means fewer interface elements shown at once. A side-effect of this is that interfaces have minimal clutter or what Edward Tufte refers to as "computer information debris."

In many GUIs, all possible activities are available to you at once. While this allows the user to transition to any task with minimal interaction or mode changes, it also risks overwhelming the user. It may seem that by making all tasks available within a click or two is powerful, this can result in choice overload.

Choice overload is a psychological phenomenon where people who are presented with many choices end up not being able to make any choice at all. In contrast, presenting only a few choices allows quicker and easier decisions. This has implications in many fields including interface design. Consider figure 1.3, which shows LingsCars.com, an actual car leasing website, on the top and BestBuy.com on the bottom. Which website style would be easier to choose what to click on?

Figure 1.3 Comparing two websites in terms of choice overload. On the top, LingsCars.com has an uncountable number of links and visual elements, many of which blink on the actual website, and visitors will take a while to process all the choices and make one. On the bottom, BestBuy.com presents only a few choices in a much cleaner fashion, allowing the visitor to make an easier choice between fewer items.

Both this guideline and the progressive learning guideline suggest limiting the number of options at any particular point. This does not mean that your natural user interface needs to have fewer overall features than the equivalent GUI application. The application can still have many features, but they should be available in the proper context. The user can use the high-frequency interactions to quickly transition from one context to another.

The direct interaction guideline is influenced by how we interact with the real-world. It may help to use a real-world example to help you solidify what this guideline is about.

REAL-WORLD DIRECT INTERACTION

To help explain direct, high-frequency, contextual interactions, suppose you are cooking a meal in the kitchen. You pull out a pot, add all the ingredients, and heat and stir. While stirring, you have many, quick

interactions with the cooking mixture in a direct way. If you want to make sure it is not burning on the bottom, you can use a spatula to scrape the bottom and mix it some more. You get feedback as you stir about the consistency and texture, and through many, small interactions you get a sense for how the meal is cooking.

Even though you have many other things you could be doing instead of stirring the pot, you do not have sensory overload because the vast majority of those other activities are performed in a different context. Doing the dishes requires being at the sink rather than the stove, and doing laundry involves clothing and washing machines, which are normally not at hand while cooking. Even though those activities are not immediately accessible, you can quickly transition to them by walking to a different area of the house.

Computer interface technology isn't quite to this level of multi-sensory interaction and feedback, but we can still emulate some of the qualities. As a point of contrast, consider how a GUI would implement a cooking activity. Using standard GUI interaction patterns, I could imagine a wizard like the one shown in figure 1.4.

Figure 1.4 GUI interfaces use indirect, low frequency interactions. In real life you could have many quick interactions as you cook and make small adjustments to produce a great meal. GUI interfaces have fewer, more significant interactions and feedback such as this wizard.

There are only a few interactions to set up the parameters, followed by a progress bar representing the length of the cooking process and occasionally status updates. At the end, the wizard reveals that there was an error in your original parameters (perhaps you cooked it for too long or didn't stir enough) and the meal is ruined. Helpfully, it offers you the option to retry. This example may seem funny because it would be a horrible way to attempt to cook a meal, but the sad part is that is the status quo with the many activities people routinely perform with graphical user interfaces.

NOTE

We are covering a lot of different concepts very quickly here, so don't feel bad if you have not fully absorbed them yet. In the next few sections and in the rest of the book, I'll be giving you concrete examples that will help you understand these concepts. Feel free to come back and re-read these sections once we've gotten through those examples. You may find that after seeing some of these ideas in action that you actually do understand what I'm talking about.

We discussed four natural interaction guidelines. While there are many different ways to approach NUI design, I believe that these four guidelines are a simple, easy-to-remember way to start your design efforts in the right direction. In the next section we will take a look at how we can apply some of the natural interaction guidelines and NUI concepts we've learned so far.

1.4 Applying natural to NUI

At this point, you are well on your way to understanding the natural user interface. It is time we start applying some of the concepts we have discussed in this chapter to a few problems you may come across when creating natural user interfaces. It is important for these core concepts, including the meaning of natural and the natural interaction guidelines, to become an integrated into your thinking processes. When we start thinking about specific technologies such as multi-touch, you will need to apply these natural human behavior concepts as if they are second-nature.

As we discussed in section 1.2, a key component of natural is reusing abilities and skills. Let's work through some examples about how abilities and skills can apply to natural user interfaces.

1.4.1 Reusing innate abilities

As we discussed, we develop our innate abilities automatically and for the most part, we use them without any conscious thought. I would like to walk through how we can use the innate ability of object permanence in a NUI. We will talk about what object permanence is and see how it can be applied to file management. From there we will talk about how this is different from how GUIs open and close files and see how this leads to some design considerations such as focusing on content and creating fluid transitions.

OBJECT PERMANENCE

Object permanence is a cognitive development milestone where a child understands that an object still exists even when it cannot be seen, heard, or touched. It is a key concept that develops during the first two years of life and is critical for the understanding that objects are separate from the self and that objects are permanent. Toddlers who understand object permanence will look for hidden objects in the last place they saw them.

This understanding is completely automatic once it matures. I do not expend any cognitive load wondering if my car will cease to exist if I park it and walk into a building where I cannot see it. I understand that I can find my car where I last left it. In a couple cases, I have returned to where I expected my car to be but it was not there. This causes a significant amount of stress. Most of the time, my memory was faulty and I forgot that I parked in a different spot. Other times, my car was moved without my knowledge by my wife, or one time a tow truck! (I was parked legally, honest.) When something is not where we expect it, it violates our understanding of object permanence and we must spend time and effort resolving the conflict between our expectations and reality.

We can apply object permanence to interface design by ensuring that we do not violate the user's natural expectations that content is permanent and remains in the location and condition they left it.

With NUI we are talking about fundamentally changing how we interact with computers, but behind the scenes we still have to deal with the same operating system architecture and computing abstractions. We will still have to deal with concepts like device drivers, processes and threads, and files and the file system for a long time. Let's talk about some of the challenges we have with file management and then we will see how we can apply object permanence to this problem to hide the complexity from the users and make their experience more natural.

File management challenges

In regular GUIs when we create content such as a Word document, we have to worry about saving the document as a file, what to name the file, and what folder to put the file in. The implementation of GUI file management has a few issues if we are trying to create a NUI. Managing files in GUIs is a composite skill that takes time to learn. Even though the file and folder metaphor has roots in real-life, there are many file and folder behaviors that are not natural. For one, if we forget to save then our content can be lost. This risk of forgetting is reduced through dialog box warning, and risk of loss is reduced in some applications through auto-saving, but often this is configured to save every five or ten minutes.

First time computer users do not understand all of the subtleties of files and the file system and they will often only learn them through frustrating data-loss experiences. With NUI we have the ability to make the experience better for both novice and experiences users. Let's discuss an imaginary application to illustrate how to use object permanence to make file management more natural. For discussion we will call it NaturalDoc.

File management with NaturalDoc

NaturalDoc is designed for use on a multi-touch tablet and lets you create a document of some type and takes care of all of the file management for you. It automatically saves the every time you make changes and automatically figures out what to name the file and where to put it in the file system. All you have to worry about is the consuming or creating the content.

This type of application behavior abstracts away the difficult to learn inner workings of the computer that are often unnecessary for the user to know. There are some existing applications such as Microsoft OneNote that follow this pattern. In OneNote, you take notes and at any time close the application. Later when you launch it again your notes are right there where you left them. OneNote is a good example of object permanence, but to really make the content seem like a permanent object it shouldn't just disappear when you close the application.

Opening and closing files

In a standard GUI application, when you close an application, the content disappears along with the application window. If you want to access it again, you have to find and double-click on a specific icon within a folder hierarchy that represents where the file is stored on the hard drive.

The GUI-style of opening and closing files is very disruptive to the concept of object permanence. In some cases your content is presented in full form, but then it disappears and is represented indirectly as a file icon somewhere else with an entirely different visual. The only connection between these two representations of your content is logical.

Another common GUI-style pattern to access your content would be to launch the application then open the file from there, either by navigating a folder hierarchy in a dialog box or by choosing from a list of recently opened files. When you launch the program, what you are really doing is telling the computer to begin executing the code that constitutes that program. Like the file system, execution of code and processes is another detail about how computer work that is not necessary for the user to know. Yet in graphical user interfaces, it is common for users to worry about whether a program is running. All they should really need to worry about is what content they want to be using at the moment.

Let's take NaturalDoc a step further and address these concerns. First we'll apply the direct interaction guideline and then apply object permanence again.

Content-centric interfaces

Forget about the application. Don't even bother with it. "Hold on," you may think, "you're getting kind of crazy on me." Actually what I'm doing is getting natural. Remember in the definition of natural user interface, a key component is "interacting directly with content." NUIs are not about interacting with applications or menus or even interfaces. The content is important. Let's use the direct interaction guideline to design NaturalDoc to be content-centric.

Content-centric means there is no application, or at least no application interface. Sure, behind the scenes there will be processes and threads for the application, but the user doesn't care. The user wants to interact with the content.

For NaturalDoc, each document the user creates or views will be a separate object. One unit of content is one object. These objects will contain the visual for the content as well as the necessary interface elements for interacting with the content. The interface elements would be as contextual as possible. You can minimize the number of interface elements by making them only appear in the appropriate context. High-frequency interactions allow the user to move quickly from one task to another, navigating the contextual interface elements and performing actions. Figure 1.5 shows how one interaction pattern for changing the font size of text and is a great example of the direct interaction guideline.

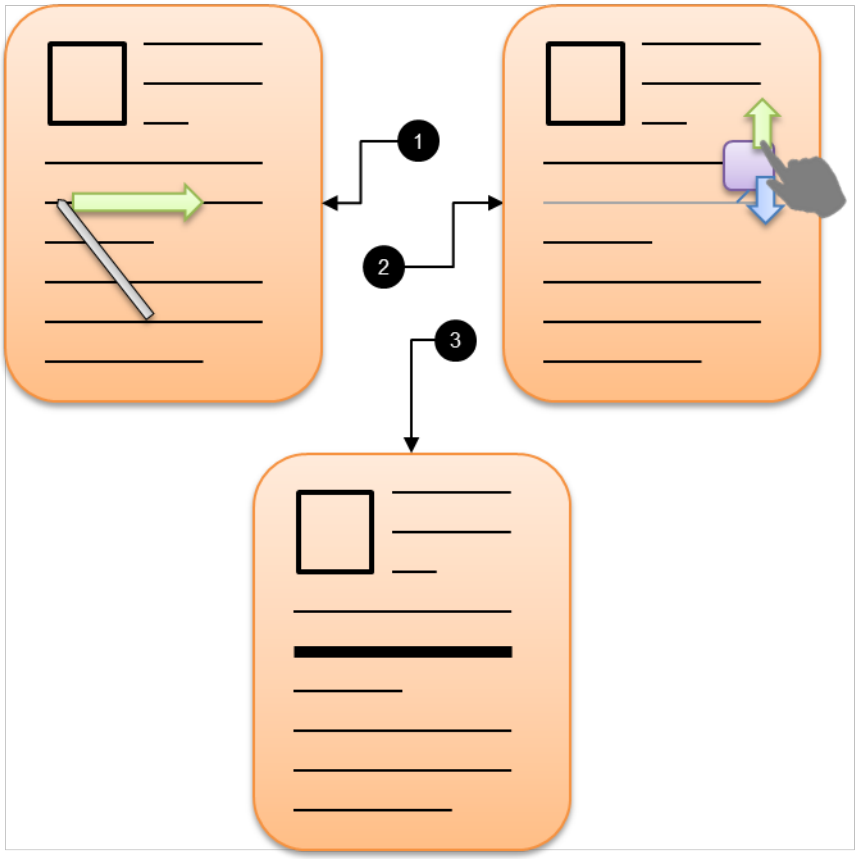


Figure 1.5 A contextual interaction for changing the font size of text. The user uses a stylus for writing and selecting text. When text is selected (1) a manipulation handle appears and can be used with a pinch manipulation (2) to change the font size (3).

Numbers in figure 1.5 and in caption and text need to be done as proper cueballs.

One way to change font size with a contextual interaction starts by dragging your finger over the desired text (1). This causes the text to highlight and a finger-sized manipulation handle appears nearby. If you use a pinch manipulation on the handle (2) then the font size of the text will change appropriately (3). The handle could also be used for other interactions such as moving the text or making other style changes.

As soon as you make this change, the application automatically saves the file in the background. We can continue with other tasks and the manipulation handle will fade away as it becomes out of context.

PERMANENT OBJECTS REQUIRE FLUID TRANSITIONS

Let's talk about one last interaction we can design if we combine the content-centric nature of NaturalDoc with object permanence. When you finish interacting with a document, instead of closing an application (since there is no user-facing application, only the content objects), you could use a pinch manipulation on

the entire document to shrink it down and put it away somewhere, illustrated in figure 1.6. Later you could come back to the same place and enlarge the content to interact with it again. This transition from active to inactive and back could also be automatically triggered with an appropriate gesture or interface element.

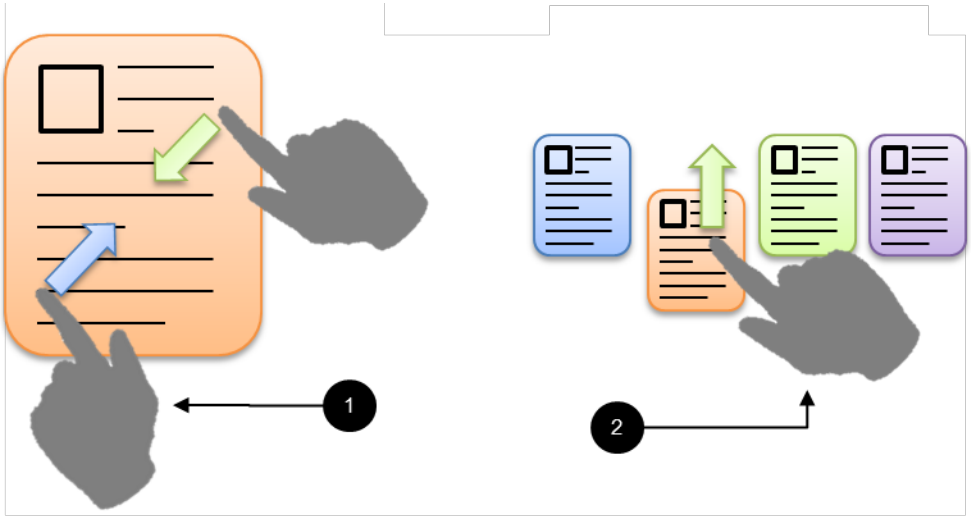


Figure 1.6 The discussion application, NaturalDoc, is organized around documents as permanent objects. These objects can be scaled up and down using pinch manipulation #1 and then stored within a spatial organization system #2.

Numbers in figure 1.6 and in caption need to be done as proper cueballs.

Shrinking and moving the document implies a spatial way to store, organize, and find inactive content. Coming from the GUI world, we might be tempted to think of this like the desktop metaphor, but there is no reason why it should be so. These documents could be contained within a zooming canvas, a linear timeline, or another interface metaphor that makes sense for the content.

This specific description of NaturalDoc may or may not be the best approach for a real application, but the important part is how we took an innate ability that all humans have and used it to guide several different design decisions in the interface and abstract away unnecessary details of how a computer actually works.

In NaturalDoc, the content is a permanent object and there is a fluid transition between different object states. Maintaining a fluid transition keeps the user oriented and maintains the suspension of disbelief. If the transition between states is not fluid, the user will not be able to intuitively understand how to go back to the previous state.

As a permanent object, NaturalDoc automatically takes care of saving files and the file management behind the scenes. This minimizes the number and complexity of skills required for novice users to get started with the application.

The best part of reusing innate abilities like object permanence is that users don't need to learn anything to understand it. The content just behaves like a real life object would behave. Not everything in an interface can be done using innate skills, though, so some learned skills are required. Let's discuss how to reuse basic skills in a natural user interface.

1.4.2 Reusing basic skills

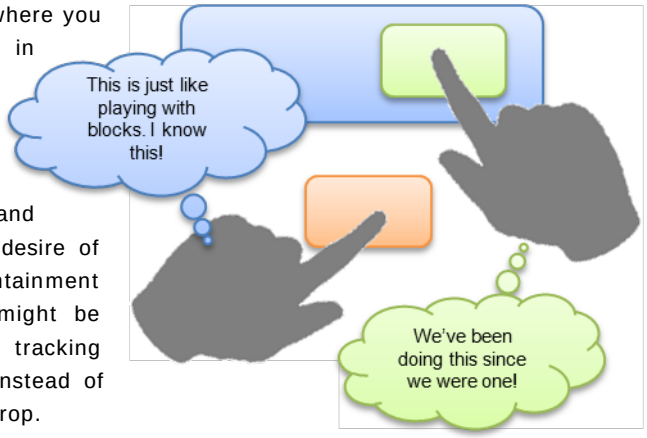
Basic skills are very easy to reuse and apply to new situations. This makes them perfect for natural user interfaces. For an example, let's use a skill that builds upon object permanence. Once toddlers understand

that things exist even when they can't be seen, they can combine that with the containment relationship and their ability to grasp and move objects to create a simple skill: putting objects in other objects.

The containment relationship is one of the first concepts we understand as infants. As soon as we are able to grasp and move objects, we start to play with the relationships between objects. Playing with and changing the containment relationship is at the core of a lot of infant and toddler self-directed play. My kids love putting toys inside of boxes or baskets, carrying them around, and then taking them out again. Sometimes I can get them to play the "put your toys away" game, which is basically putting the toys inside the appropriate containers.

We can apply our container skills to NUIs in many different situations. Let's see how they can be applied for categorizing items and compare that to how we can categorize items with GUIs.

CATEGORIZING ITEMS



Imagine an activity in a NUI application where you categorize objects by placing them in containers. This would be a very natural and easy task. We intuitively recognize the containment relationship of the various objects and containers. Once we figure out that we can interact with and move the objects, we can achieve our desire of categorizing objects by changing the containment relationship. On a touch screen, this might be called drag-and-drop. With a motion tracking system, we might use a grab gesture instead of touching, so it might be called grab-and-drop.

There are several options within GUI application for categorizing items. Here are a few interface patterns, in order from most cognitive load to least cognitive load:

1. List of items with checkboxes and a central category selector and apply button
2. List of items with dropdown lists of categories.
3. Two lists side-by-side with left and right arrows to change item membership
4. Drag icons from one area and drop in another area

The first three of these GUI categorization patterns requires significant learning to use the controls as well as the interaction pattern. The extraneous cognitive load is also pretty high. The last pattern is almost the same as the NUI drag-and-drop method we just talked about and is the most direct and easiest to understand. Unfortunately, it is rarely used, except for moving files from one folder to another.

Playing with containers is something we all know, but there are more complicated computer activities. Sometimes not everyone has learned a skill. Sometimes we will have to learn something new for a novel activity where no acceptable existing skill exists.

1.4.3 Teaching new skills

Let's say you have identified an interface task that you cannot satisfy using an existing skill common to most humans. As you add more advanced features to your application, this will undoubtedly happen. Let's discuss how to teach a simple skill that is natural but many users many may not know beforehand: the two-finger pinch.

The two-finger pinch can be used as a manipulation or a gesture. (We will discuss the difference between manipulations and gestures in chapter 2.) In natural user interfaces, it is commonly used for resizing objects. The cliché example is resizing photos. Pinching can also be used for task that requires changing a numeric value. In section 1.4.1, I described how NaturalDoc could use a pinch manipulation to change the font size of text.

Table 1.2 shows the two things that the user needs to learn for the font-resize task.

Table 1.2 Some advanced interface tasks require learning. The font-resize task requires a skill that is composed of two learned items.

Learned item	Description
The pinch skill	This is purely the skill of using touching the screen with two fingers and moving them relative to each other. As a simple skill, it is easily reused for many different tasks in different situations.
The font resize interface behavior	This interface behavior is really the association between pinching a

particular visual element and changing the font size. Interface behaviors are often understood in terms of metaphors, which will be covered in detail in chapter 2.

In this case, the pinch skill is something that could potentially be used in many different situations. In NaturalDoc we also used pinching to shrink down the active document and put it away as well as for enlarging inactive documents to start interacting with them.

OBSERVING SKILLS SOCIALLY

There are several different ways to learn an interaction skill. If it is a simple skill, then the easiest way is to observe someone else executing the skill. Once when my oldest daughter was three, she watched me test a new multi-touch map application. As you can imagine, this application used pinch manipulations for zooming in and out of the map. After she watched for only a minute, I asked her to try. With minimal observation and no practice, she was able to perform the same pinch manipulations that I was doing. Considering her still-developing hand-eye coordination, I was impressed.

For applications that are used is a social setting, observation is a good way to learn how to operate the interface, but it cannot be depended upon alone. Sometimes no one is around to show the new user how the interface works.

DEMONSTRATING SKILLS DIRECTLY

One way to address this is to have the application itself demonstrate a skill. On a multi-touch device, this could be done by showing an outline of hands interacting with the interface. Alternately, a video or animation of someone interacting with the interface could be displayed. This is an approach that some video games take, notably on the Nintendo Wii. On the Wii, the player must move the controller in specific way for certain interactions. If the player is not performing well, some games display a cartoon in the corner showing a character using the controller in the proper way. While this type of teaching works well for a video game, I do not think it is the best approach for general-purpose natural user interfaces.

One of the problems with this is determining when to play the demonstration video or animation. Videos games can generally measure the progress of a player against some metric, but this is hard or impossible in a regular application. Some applications will trigger the demonstration animation if the user has not touched the application for a period of time. Figure 1.7 shows a screenshot of the Microsoft Surface Financial Services sample application, which has this feature. The problem with that approach is that the lack of interaction might not have anything to do with being stuck. The user could simply be doing something else in real life, or could be talking to someone else about what is currently displayed.

Figure 1.7 The Microsoft Surface Financial Services sample application demonstrates to the user how to interact with it by displaying hand silhouettes and playing an animation. This animation showed how to navigate from an activity overview mode to a specific activity by using a pinch manipulation to zoom the interface. There was no visual affordance that the user could do the pinch manipulation. Instead, if the user did not interact for a certain length of time the hands would appear and act out the action the user is supposed to perform.

Another problem with a demonstration video or animation is that users may feel somewhat insulted that the application thinks they do not know what they are doing. The user may feel the application is taking control away or is disrupting something else the user was doing. The context is different than a video game, where players know clearly if they are doing badly and probably want help.

In almost all cases, there are ways to intelligently design your application to subtly teach the user what they need to know. If the user cannot figure out how to perform a task by playing with the interface and you need to resort to an animated demonstration, then you probably need to rethink the interaction design of that task.

The best approach is to apply the progressive learning guideline and create learning tasks.

1.4.4 Learning tasks

Learning tasks are regular tasks that are enhanced to teach a specific skill or interaction pattern. Good candidates for learning tasks are tasks that are performed early on by new users. These tasks do not need to be exclusively dedicated to teaching. They can serve useful purposes in addition to teaching.

Learning tasks should be worked into the flow of an application to make sure the user learns the important skills such as pinching before they are needed for other tasks such as changing the font size.

SURFACECUBE

To help explain the learning tasks, I'm going to discuss a Microsoft Surface application I created called SurfaceCube for the purpose of demonstrating good usability in natural user interfaces. SurfaceCube is basically a 2x2 Rubik's Cube game. It has a 3-D cube and you can spin the cube using a pan manipulation or rotate the layers of the cube using a hold-and-flick gesture. The hold-and-flick gesture involves putting one finger on one cubelet that you want to remain stationary and flicking another finger on a different cubelet that you want to rotate. Figure 1.8 shows me interacting with the cube on Microsoft Surface.

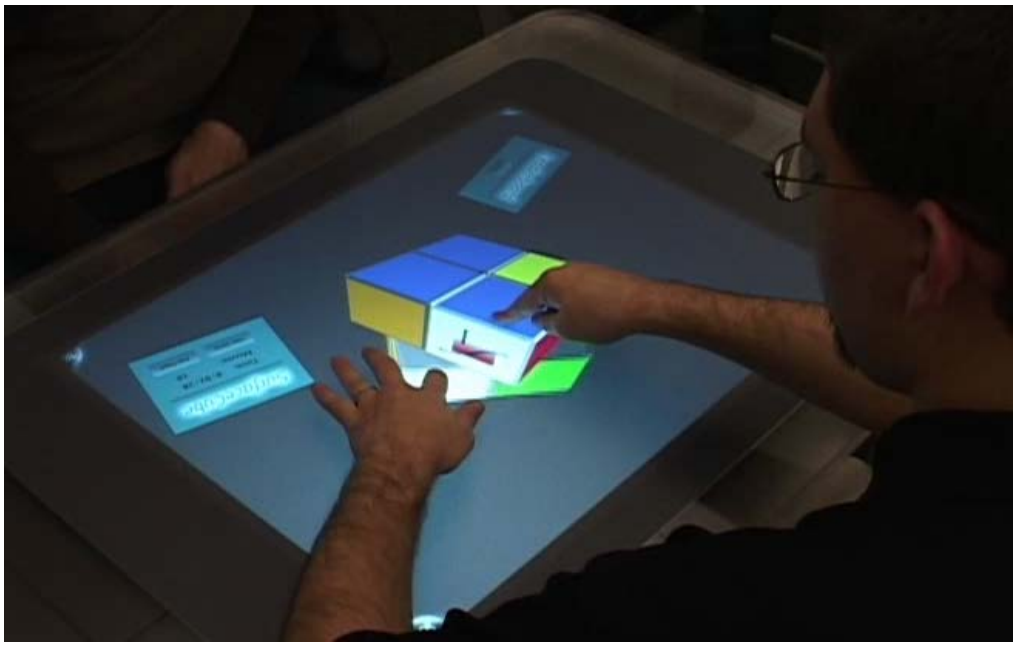


Figure 1.8 Interacting with SurfaceCube on a Microsoft Surface device. SurfaceCube is a Rubik's cube style game designed to illustrate good usability and interaction design in natural user interfaces. Here I am showing how to use a hold (right finger) and flick (left finger) gesture to rotate a side of the cube.

One of the challenges in designing the application is how to teach the users how to interact with this 3-D object. To do this I changed how the user started the game. When the application starts, the user is presented with two ScatterViewItems. One item is a leader board and the other is the main control panel that the user can use to select the difficulty and start the game.

NOTE

ScatterView is a WPF panel in the Microsoft Surface SDK that enables content to pan, scale, and rotate freely in response to touch manipulations. Content within a ScatterView is contained within ScatterViewItems. ScatterView is made available for WPF 4 with the Surface Toolkit for Windows Touch, discussed in chapter 5.

Instead of using a button labeled "Start", I decided to make this task of starting the game into a learning task. There were two important things I wanted to teach:

- You can spin the cube using a pan manipulation
- You can rotate a side using a hold-and-flick gesture

I decided that in order to start the game and get to the main cube, the user must perform a hold-and-flick gesture on cube in the control panel. I didn't want all of the complexity and degrees of freedom of the main cube, though, so I created a mini-cube with only two cubelets. The ScatterViewItem with the mini-cube is shown in figure 1.9.

Figure 1.9 SurfaceCube control panel in the pre-game configuration showing the mini-cube. The mini-cube is a simplified version of the main game cube intended to help teach users how to interact. Users must solve the mini-cube using a hold-and-flick gesture in order to start the game.

This mini-cube could spin only on the pitch axis, unlike the main cube that could spin on the pitch and yaw axes. Also, only having two cubelets, it could only be rotated in one way and only had four possible configurations. The starting configuration of the mini-cube was one hold-and-flick from completion. Table 1.3 show the four different techniques I used to subtly teach the user what the mini-cube was capable of and how they should interact with it.

Table 1.3 The mini-cube task was designed to teach the user the skill of interacting with a 3-D cube. It uses several subtle and redundant techniques to ensure that the wide variety of users will be able to successfully learn the skill and perform the learning task.

Technique	Implementation	Description
Simplified task	Two cublets with one rotation axis	While the main cube has eight cubelets and has two rotation axis, the mini-cube is simpler. This reduces the number of variables and unknowns the user has to think about and increases the chance they will accomplish the task.
Show what is possible	Timed animation	If the mini-cube is not touched, every twenty seconds it will perform a three step animation that lasts approximately one second. The animation shows the range of motion possible with the mini-cube.
Iconography	Fingerprint icons	The initial front-facing sides have icons on them. The red face has a single fingerprint, and the blue face has a fingerprint with a down arrow. The fingerprints hint where the user should touch and the arrow on one face hints at motion for one side only.
Visual tension	"Start" text on separated red faces	The initial upwards-facing but visible red face has the text "Start" on it. This text and the tension caused by the almost solved cube links the idea that resolving the tension by solving the mini-cube will lead to starting the game.
Text prompt	Textual instructions	As further reinforcement, below the mini-cube is the text "Match colors to start."

The timed animation, shown in detail in Figure 1.10, is different than the Financial Services sample application demonstration animation we discussed in section 1.4.3. This animation did not use hand silhouettes for demonstrating the actual task. Instead it is simply draws attention to show what movements the mini-cube is capable of, but is subtle enough to not be a distraction from the rest of the interface.

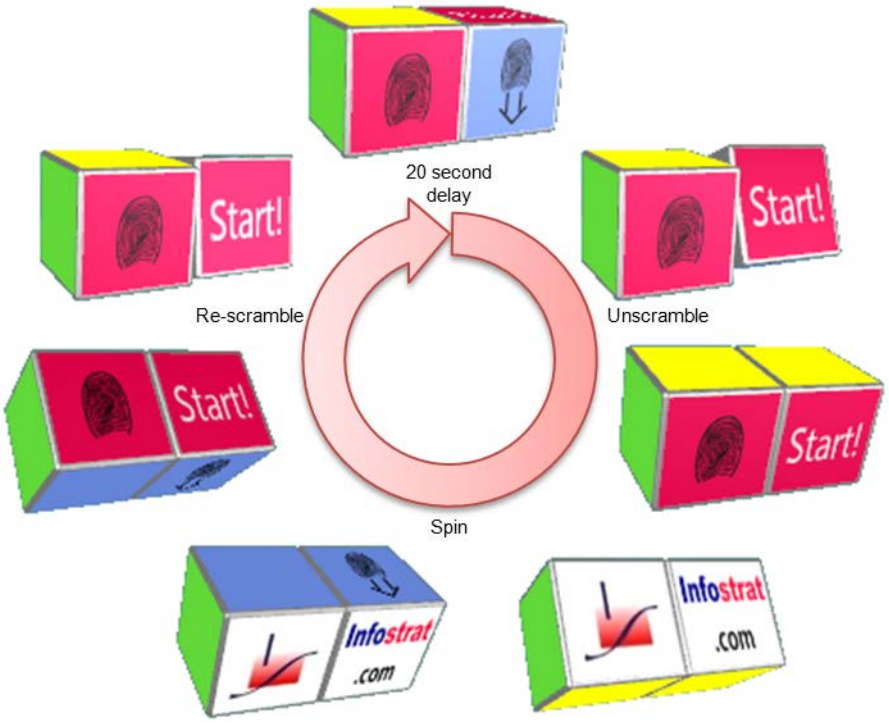


Figure 1.10 When the user is not interacting with the mini-cube, it executes this one second animation sequence every twenty seconds. The animation shows the mini-cube solving itself, spinning around exactly once, and then returning to the initial unsolved position.

Once the user solves the mini-cube, the interface rearranges itself to make room for the main cube. The main cube then fades it in the solved configuration, then scrambles itself a certain number of times depending upon the difficulty chosen. Showing the scrambling serves the same purpose as the mini-cube animation and gives the users clues to what they can do with the main cube.

Putting all of the mini-cube features together, it subtly teaches the user how to interact with the game without being distracting or in the way of expert users. The requirement that the user performs the hold-and-flick gesture ensures that the user knows in principle what to do when they get to the main game.

The mini-cube is a great example of how to implement a learning task and how the progressive learning guideline can be applied. We also saw examples of how reuse existing simple skills and how reusing innate abilities can guide us through design problems. As we start to discuss interface metaphors in chapter 2 and start to actually write some code later, we will continue to apply the natural interaction guidelines.

1.5 Summary

Congratulations! You've made it through this chapter! I know it was very conceptual and I covered a lot of theory, but these are very important ideas that will make a huge difference in the kinds of multi-touch applications you create. You have completed the first step in participating in the natural user interface revolution.

We have discussed what the natural user interface is and how you can reuse existing skills to create natural interactions. We have seen how cognitive load can be minimized by favoring simple skills over composite skills. I have armed you with four natural interface guidelines that you can use to help guide your thoughts and designs.

By now you should be realizing that creating natural user interfaces requires a whole new way of thinking. NUI is inspired by an understanding of human cognition and focuses on how humans learn and

naturally interact with our environment. If you want to create really high-quality natural user interfaces, you will need to embrace this way of thinking and resist temptation to revert to legacy GUI patterns.

A WORD OF CAUTION

We can learn from GUI and reuse things that work well, but you need to be cautious about the terminology you use when talking about NUI. We have to not only think in terms of human interaction but also speak in terms of human behavior. It would be easy to say click when we really mean tap or grasp. Someone else who has not done the training and research that you have will hear you say click and all of the GUI conceptual baggage will be applied. They may start thinking about cursors and right-click menus when none of that is appropriate.

As a participant in the natural user interface revolution, you must also be a NUI ambassador. Watch out for people who have confused or mixed ideas about what NUI means and help them understand. You may come across someone who thinks that multi-touch technology is a failure because their GUI does not use it to its full potential. GUI was designed for the mouse, so GUI plus touch does not equal NUI. Of course they probably had a bad experience, and it is up to us to help them see the potential of a NUI future.

EVOLUTION OR REVOLUTION

It is true than NUI will not suddenly replace all GUIs overnight, as the word revolution might imply. The input technologies necessary for NUIs will gradually gain market-share and the adoption of natural user interfaces will occur on the same pace. In that sense, NUI adoption is an evolutionary process and will slowly replace GUIs for general purpose computing tasks.

On the other hand, the new way of thinking required for NUIs is not an evolution of GUI at all. Your personal transition from GUI-thinking to NUI-thinking will be filled with "ah-ha" moments, "eureka!" moments, and moments of clarity when you forever think of computing in a different way. This is why I call it the natural user interface revolution.

YOUR NEXT STEPS

In the next chapter, we will continue drawing from human cognition and discuss a set of metaphors that can be used for organizing human-computer interaction with NUIs. We will use those metaphors to organize all of the interfaces we create for the rest of the book. In chapter 4 we will start learning the WPF 4 Touch API and then you will be creating your own NUIs.

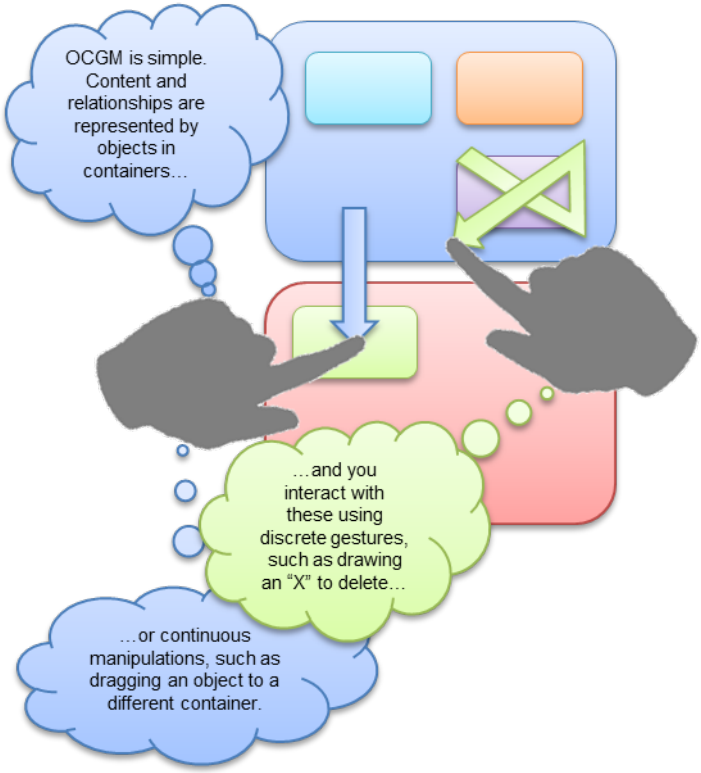
Welcome to the natural user interface revolution! I can't wait to see what you will come up with.

2

Understanding OCGM: Objects and Containers

In chapter 1, we spent a lot of time discussing skills and we also discussed some guidelines for creating natural interactions. Those concepts are core to understanding natural user interfaces, but they are only the first step.

At this point, you might find it to be a challenge if you tried to apply what you know right now to create a NUI. Despite knowing that you should reuse skills and focus on basic skills, you are probably wondering how to apply this knowledge to create a practical interface. To address this problem, I would like to introduce a set of concepts that provide a solid foundation for building NUIs: OCGM, which stands for objects, containers, gestures, and manipulations.



The concepts behind OCGM are definitely useful to designers and user experience specialists, but they are also useful to developers and I would encourage developers to read and understand these ideas as well. Everyone learning about OCGM would help a NUI project because the designers and developers would use the same terminology and understanding the same core concepts. OCGM can also help developers digest complex project requirements into building blocks of NUI functionality. Table 2.1 provides a summary of OCGM.

Table 2.1 OCGM stands for objects, containers, gestures, and manipulations. These are the foundational concepts necessary for designing NUIs.

Concept	Definition	Example
Objects	Units of content or data	Image, Document, Data record
Containers	Represents the relationships between content	List of objects organized by category or name
Gestures	Discrete interaction (direct or indirect) based upon human-human nonverbal communication.	Tapping or flicking an object, drawing an editing mark over text
Manipulations	Continuous, direct interaction based upon human-environmental interaction	Drag and pinch an image into position in a document

In this chapter we will focus on high level OCGM concepts and details of objects and containers. In chapter 3 we'll learn about gestures and manipulations. Each of these concepts in OCGM are metaphors. Let's set the stage for discussing the OCGM metaphors by briefly talking about the role of metaphor in interfaces and how they help the user understand interface behavior.

METAPHORS AND BEHAVIOR

Behavior is how something acts and reacts in a particular context. The human user and the computer interface each have separate behaviors. Figure 2.1 shows an example interaction that illustrates action and reaction of human and interface behaviors. You can help the user to understand and predict the behavior of your NUI using perceived affordance and metaphor.

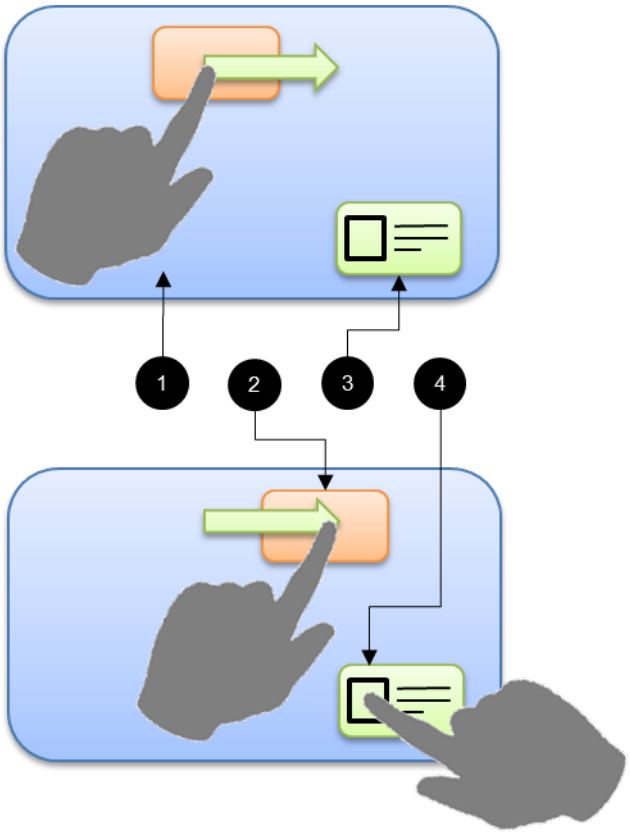


Figure 2.1 Interaction behaviors are composed of actions and reactions. In this example, the user action is dragging a finger across a touchscreen #1 and the interface reaction is moving an interface element so it tracks under the finger #2. The interface can also initiate actions such as displaying a notification #3 and the user can react by opening the notification #4.

Perceived affordance is a visual quality of objects that suggests how they can be used. For example, a button looks like it can be pushed if the button's visual appearance has particular gradients and shading to make it look raised from the surrounding area. Visual design of affordances can only explain small behaviors of individual interface elements. You need to also help the user understand how they can interact with multiple interface elements, navigate from one mode to another mode, and progress towards accomplishing tasks. The user can learn and predict these high-level behaviors if you use another technique: the metaphor.

Metaphors are used in language and interfaces to describe one thing in terms of another. When Shakespeare wrote "All the world's a stage" he applied attributes of a stage to the world in general. In interfaces, the desktop metaphor applies attributes of real-world desks and common office objects to an interface. In both cases we understand the complex thing better by thinking about it in terms of something else. Metaphors help us understand the high-level behavior of interfaces.

Conceptual metaphors

Metaphors are used in literature and interfaces, but they also play an important role in human cognition and thought. Conceptual metaphors are cognitive patterns that operate on a more basic level than language and help us understand complex concepts in terms of more familiar concepts.

Conceptual metaphors build layers of concepts and understanding by mapping concrete sensory experiences to more simple concepts, and mapping from simple concepts to more complex concepts.

The complex concepts are called the target domain, and the familiar topics are called the source domain. Metaphors map attributes and patterns of the source domain onto the target domain, as seen in figure 2.2. This allows us to understand the target domain in a new way while hiding some complexity.

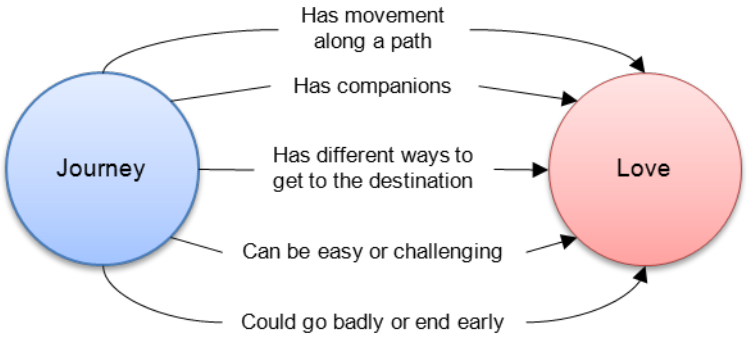
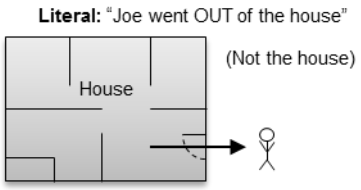


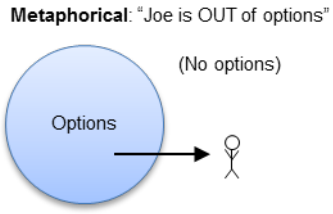
Figure 2.2 Conceptual metaphors map attributes of a source domain onto a target domain. The metaphor "love is a journey" maps attributes of the concept of a journey onto the more abstract concept of love. This enables us to understand phrases such as "our relationship isn't going anywhere", "I want to get us back on track", "it's been a long, bumpy road", and "we should go our separate ways".

As an example, consider an infant that observes and interacts with her environment and develops an understanding of basic spatial relationships such as CONTAINER, IN, and OUT. These low-level understandings are called image-schemas, and they can be used in a literal sense or as source domains for conceptual metaphors. Names of image-schemas and metaphor mappings are conventionally written in all upper-case letters.

When used literally, such as in the phrase "Joe went out of the house", the OUT image-schema describes a transition from inside to outside of a boundary.



When used metaphorically, the OUT image-schema can help us understand other types of transitions across a metaphorical boundary. For example, the phrase "Joe is out of options" implies a transition from a state where there were options to the new state where there are no options. Instead of saying "Joe has no more options," we can use the word "out" metaphorically. This causes us to construct a physical space in our minds where there are options and a boundary between that space and somewhere else that has no options. Joe then makes a metaphorically physical transition from the options-space to the no-options-space.



The OUT image schema is responsible for us understanding this spatial transition. When we use it in a non-spatial way, our minds automatically construct the necessary conceptual metaphor and domain mappings so we can make sense of the phrase.

When we design computer interfaces, we can draw from the user's experience and existing conceptual metaphors, from basic spatial metaphors to complex experience metaphors, to help the user understand the behavior of the interface.

Each generation of interfaces has a different set of common behaviors and unique metaphors to explain those behaviors. GUIs use the metaphors of windows, icons, menus, and pointing devices, or WIMP, to explain the standard behaviors of GUI applications. While there is a lot of variation in GUIs, the WIMP metaphors are persistent.

In NUIs, the behaviors described by the WIMP metaphors are not useful. The WIMP interface behaviors require composite skills and significant learning to master. NUI requires new behaviors and new metaphors. OCGM describes the core metaphors for NUI: objects, containers, gestures, and manipulations.

2.1 OCGM's Razor

OCGM promotes the natural human behaviors that we want to encourage in NUI by reusing common human skills. The acronym, OCGM, can be pronounced Occam, as in Occam's razor. This is a beautiful double meaning because it fits nicely with the philosophy of NUI.

Occam's Razor

Occam's Razor is a philosophical principle that can be used as a heuristic for evaluating theories or hypothesis. It says that entities should not be multiplied unnecessarily, which can be interpreted to mean that the theory with the fewest unknowns or variables is preferred.

When discussing Occam's Razor, care should be taken not to apply it as a logical certainty or an absolute rule. It is not meant to be used as a final arbiter between theories. Instead, it is meant to help guide the development of theories and promote explanations that have the fewest number of unknowns.

The spirit of Occam's Razor is embedded into the natural interaction guidelines discussed in chapter 1. Given a task of any particular complexity, the interaction that minimizes the cognitive load and requires fewer or simpler skills is more likely to be the right interaction for the task.

This guidance should not be applied in a vacuum, though. Interfaces created using the natural interaction guidelines still need be verified through usability testing or other processes. By using those guidelines and the spirit of Occam's Razor, you are more likely to have positive results during testing.

Like WIMP, OCGM are metaphors that can help you and your users understand the core behaviors of NUIs. Let's talk about what the OCGM metaphors are and the benefits of using OCGM.

2.1.1 OCGM and metaphor

The OCGM metaphors can be categorized into four categories, shown in Figure 2.3.

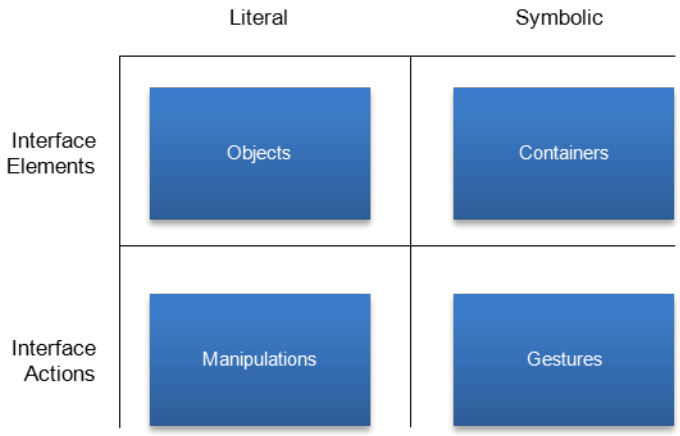


Figure 2.3 Objects, containers, gestures, and manipulations metaphors can be categorized by whether they are literal or symbolic and whether they are interface elements or interface actions.

Objects and containers are types of interface elements. They are presented by the application using whatever output technology is appropriate, whether graphics, sound or voice, tactile, or future output

technologies such as 3-D holograph. Gestures and manipulations are types of interface actions that the user performs. The user provides input to the application using whatever input technology is appropriate, including multi-touch, motion tracking, stylus, voice, or future input technologies such as muscle sensors or brainwave sensors.

Slicing OCGM a different way, containers and gestures can be thought of as symbolic. A container is an interface element that presents symbolic information about the relationship between content. A gesture is an interface action that has a symbolic meaning. In contrast, objects and manipulations are literal. An object is literally content, with as few non-content interface elements as possible. A manipulation is a very literal action because the action and the interface response should have parallel action and generally be as direct as possible.

Origin of OCGM

The concepts behind OCGM originated from research done by interaction designer Ron George. In 2009, Ron did a broad survey of interaction designs across different types of devices and interfaces, including some GUI and some NUI. He observed some interesting patterns that led him to think about objects, containers, gestures, and manipulations as core elements of any interaction.

I became friends with Ron while I was doing research for this book and talked to him about some of my early research and original ideas regarding the post-WIMP metaphors. He shared his thoughts about objects, containers, gestures, and manipulations with me, and I was immediately hooked.

At the time, Ron did not have a pronounceable or memorable acronym for these ideas, so I wrote down all possible permutations and found one that jumped out as an obvious and elegant answer: OCGM. This specific order is memorable because it could be pronounced Occam, and thus could be linked to the very appropriate Occam's Razor.

Ron and I wrote on our blogs about OCGM and received great feedback. We presented a position paper at the CHI'10 conference NUI workshop, where Ron took the first author position as the idea guy and I took the last author position as the main author of the paper. The core thesis of the paper was that OCGM are the universal foundational metaphors for natural user interfaces. This assertion was supported by linking OCGM to several human cognition theories and analyzing it in the context of early childhood cognitive development. My ideas about NUI presented in chapter 1 and about OCGM in this chapter expand upon that paper and are also derived from human cognition research.

OCGM can be thought of as universal foundational metaphors for any NUI interaction. OCGM are metaphors because they map attributes from natural human behaviors to human-computer interaction design. They are foundational because they describe basic interface behaviors that you can build more concrete metaphors on. They are universal because they apply across many diverse form factors, such as desktop, mobile, or TV, and many different input modalities, including multi-touch, voice, motion tracking, and even mouse.

2.1.2 Benefits of OCGM

Compared to the WIMP metaphors, OCGM is more abstract, but that is part of the power of OCGM. It gives you a baseline for behavior but frees you to build any specific concrete metaphor upon it. OCGM frees you from the conceptual baggage of GUIs and WIMP. You can create interfaces that are appropriate for the specific content, task, form factor, and context using the OCGM base behavior and your own design for the specific behaviors.

NOTE

Because OCGM is abstract and applies to many different devices, you could even talk about WIMP-style interfaces in terms of OCGM. Be careful if you do, though, because unless you explain things very carefully you risk confusing people and making them permanently link icons to objects and windows to containers. While that might be a justified comparison, we need to make sure that we discuss OCGM on its own and fully develop the ideas so that people understand the benefits of an OCGM-style interface.

Another benefit of using OCGM is that users automatically use behaviors that are based upon some of the earliest concepts and innate abilities that we develop as infants and toddlers. OCGM satisfies each of the natural interaction guidelines discussed in section 1.3. Properly implemented, OCGM can help you create very natural and engaging interactions that can create a sense of flow with users.

While watching a really engaging movie, playing a challenging but well-designed video game, or performing creative work tasks, there are opportunities to enter a state of flow. In this state, we lose a sense of self-awareness and time and our minds are fully immersed in the activity itself. Flow occurs when there is a balance between our capabilities and the challenge of tasks, there is a sense of control, and there is immediate feedback on the performance of the task.

With OCGM, it is easy to create NUIs that enable your users to flow. The less your users think about the process and mechanics of the interaction, the more they can focus on the content. If you build your own concrete metaphors on top of the OCGM metaphors, you can make the most of your users' existing capabilities and the let them focus on the challenge of the task at hand, which is the perfect conditions for entering a state of flow.

The key to OCGM is how the metaphors map natural human behaviors and existing skills onto objects, containers, gestures, and manipulations. Let's take a look at the object metaphor in detail.

2.2 The object metaphor

Objects are metaphors for content. Each piece of content presented in the interface should be represented by an individual, self-contained object. An object has a one-to-one relationship with the content. Multiple instances of content should never be represented by a single object. Containers take care of organizing all of the content. Let's look at the objects metaphor in detail in table 2.2.

Table 2.2 Objects are metaphors that help us understand the basic behavior of content in NUIs.

Metaphor	Source domain	Target domain	Mapping	Example
Objects	Real-world objects	Content behavior	Content is permanent, physical, and has fluid transitions	Images that can move, rotate, scale, and be thrown

As infants and toddlers, we spent a lot of our time interacting with the environment just to learn how it worked. Once our brains and bodies developed to a certain point, we achieve the object permanence developmental milestone. Suddenly, the world made a lot more sense as we understand that the moving patches of bright colors we saw were permanent objects, not just interesting visual phenomenon.

NOTE

We discussed object permanence in section 1.4.1 as an example of an innate ability that can be used in NUI. Here we are applying the same ideas in a different way in the context of the object metaphor.

The understanding of object permanence is closely linked to the OBJECT image-schema and our developing understanding of the physical nature of objects. These cognitive developments are required for almost every single task we do as adults. Because we understand real-world objects on such a deep level, it makes sense that in NUIs content is presented using objects as the core metaphor. The object metaphor maps three primary attributes of real-world objects into the interface: permanent, physical, and fluid.

2.2.1 Permanent objects

One of the key attributes of the object metaphor is that objects are permanent. This is derived directly from the concept of object permanence. Permanent means the object should not disappear, even when the user is not looking at or using them. If the user leaves an object in a particular state at a particular position, transitions to another task and comes back later, that object should still be in the same state and position the user left it.

It might be necessary to change the object while the user is away. This may happen if the content represented by the object has a long running process, or if there were limited system resources to maintain the full fidelity of the object. In no case should the content itself be at risk. If it is necessary to change the content, special care should be taken to make the user aware that something occurred while he or she was away.

Suppose you are renting an apartment. After leaving the apartment and coming back, you expect that all of your belongings are where you left them. The landlord's maintenance crew may need to enter while you are away to fix something. In most cases, they will leave your personal items alone, or when necessary, move them with the respect necessary for handling someone else's property. When they are done, they will often leave a notice that makes it apparent that they were there. When you return, you see the notice and are put at ease that even if your items are in a different spot than you left them, it is due to maintenance rather than an unwanted visitor or burglar.

Back to computers, your user's content is their personal items. Let's design content to have a permanent presence as objects and then respect the users by leaving the objects alone. This will give the user a sense that the objects are more real and help them connect with the interface.

One important aspect of object permanence is that the objects do not exist alone in a void. They are embedded within a world. This brings us to the physical nature of objects.

2.2.2 Physical objects

In order for an object to be physical, it must exist within an environment and have physical attributes. Physical attributes determine how we interact with objects, how objects interact with each other, and how objects behave within the environment outside of interaction. The environment, whether real-world or virtual, provides context for primary physical attributes like position, orientation, and size, and enables us to perceive relative differences in physical attributes from one object to another. With OCGM, containers serve as the environment for objects.

> **NOTE**
>
> Applying physical attributes may only be possible in NUI that uses a visual output medium. It may be difficult to apply physical attributes to objects in a NUI that is voice-only with no display.

Table 2.3 lists the most common physical attributes that you may choose to implement with the object metaphor. These are some that are used in everyday real-world interactions and are innately understood, but there may be others that you might want to use as well. I categorized them into three types: object-user, object-environment, and object-object. Object-user attributes are those that the user can directly interact with, or manipulate. Object-environment attributes are those that determine how the object behaves within the environment independent of interaction. Object-object attributes determine how an object interacts with other objects.

Table 2.3 You can give objects a sense of physicality by designing the behavior to include some of the common physical attributes in this table.

Physical attribute	Type	Description
Position	Object-user	Where an object is located
Orientation	Object-user	The rotation of an object relative to a fixed axis
Size	Object-user	The total length of the object in each axis
Velocity	Object-user	The speed at which position, orientation, and size is changing
Inertia	Object-environment	The behavior when a manipulation ends. With no inertia, velocity is reset to zero. With inertia, the velocity continues to change position, orientation, and size.

Acceleration	Object-environment	How quickly velocity increases or decreases over time outside of a manipulation.
Elasticity	Object-environment	How much an object deforms when it reaches its limits
Collision	Object-object	The behavior when two objects intersect. With no collisions, the objects ignore each other and can occupy the same space. With collisions, the objects cannot occupy the same space and either merge or bounce off each other depending upon the elasticity of the objects.
Joint	Object-object	A fixed or rotating connection between two or more objects

I classified velocity as an object-user attribute even though it can be derived from position because in some cases the primary intent of the user is to set an object in motion rather than move it to a specific position. For example, if I want to get an object out of the way, I may just flick it away rather than specifically place it somewhere else. This distinction is also important once we discuss gestures and manipulations because the difference between the two is sometimes determined by whether an interaction crosses a velocity threshold.

In contrast, I classified acceleration as an object-environment attribute because it is rare and unintuitive for a gesture or manipulation to use an acceleration threshold. We understand acceleration when observed visually or by our sense of balance, but we do not have the physiological feedback necessary to create a controlled acceleration with our bodies.

PHYSICS À LA CARTE

An object does not necessarily have to look or behave like a real-world object to be physical. Unless you are creating a physics simulation, you can pick and choose which attributes to implement and how to implement them. The goal is not to model the real-world, but rather to reuse our innate ability to understand the physicality of objects for creating interactions. Our brains are wired to recognize and understand these physical behaviors, even if only some of them are present.

You may only want implement some of these attributes depending upon the metaphors that you build upon OCGM and the general layout of the other objects and containers. A particular object may only need to have a variable position and size but fixed orientation. Some attributes such as inertia and acceleration can only be observed when the position, orientation, or size is changing.

Suppose a user touches an object with a finger, starts moving it, and then lifts the finger while still moving. If inertia is implemented, the object would continue moving at the same speed as it was when the finger was lifted. Acceleration (or more exactly, a deceleration) can be used with inertia to create friction. This will cause the object to decelerate until it stops.

SUPER-REALISM

In addition to picking and choosing which physical attributes to implement, you can also play with the parameters and combine them in ways that are not possible in the real world. We have innate abilities for understand each physical attributes individually, which means we can still understand the pattern of behavior when it is extended beyond what is possible in the real world. This is called super-realism and it highlights the fact that we are taking advantage of our brain's wiring, not creating a simulation of the real-world.

> **NOTE**
>
> One of the Microsoft Surface Interaction Principles is super-realism. The intent of the Microsoft Surface principle and the concept in this section are the essentially same. I have based this guidance upon the cognitive development theme so that you understand how we understand physical attributes as individual innate abilities that can be mixed and matched in super-realistic but still intuitively physical ways.

The idea that an object can have a variable size or length uses the idea of super-realism. In our real-world interactions, there are some objects that we can change in size, but we don't necessarily understand the mechanism. In the case of a rubber band, the material is elastic and can stretch. In the case of a kitchen drawer or a retracting measuring tape, there is a mechanism that allows it to change apparent length. Of course, the materials of a drawer or measuring tape are not actually stretching, but that is the visual effect that occurs.

In any case, our interaction with these variable size objects is the same: hold two parts of the object and pull them apart or push them together. In some cases, such as the kitchen drawer, one part is secured already so you only need to grab the drawer. When we create objects, they can be super-realistic by using this same interaction to change the length or total size, even if there is no real-world mechanism that would allow this behavior.

The most common example of super-realism in a multi-touch interface is using a pinch manipulation to resize photos. We understand that like a rubber band, we have to touch the photo in two places to stretch it out. Even though there is no direct analog to this interaction in the real-world, once users understand that they can use the pinch manipulation, this interaction is very intuitive.

Another physical attribute that can be made super-real is acceleration. With inertia, the environment can have friction that causes a negative acceleration to slow down objects. You could also create a situation where the environment causes a positive acceleration to speed up objects. You might want to do this when the user triggers an action on an object and then the object needs to transition to another place or state.

To show how we can use super-realistic resizing and acceleration in object transitions, let's bring back NaturalDoc from chapter 1.

SUPER-REALISM IN NATURALDOC

In NaturalDoc, we have an object that represents a document and could also have another object that represents a document storage location. The storage object has a document container. (We will discuss containers in detail in section 2.3.) When we first discussed NaturalDoc, we talked about how the user could use a pinch manipulation to shrink the document and then manually put it away. This shrinking to an inactive state is one example of super-realism. We also said that the transition from active to inactive and back could also be automatically triggered by an appropriate gesture or interface element. Let's discuss this triggered transition in more detail.

Imagine the user is done working with a document. The user interacts with an interface element, such as an "X" button, or performs a gesture on the document, such as a flick, that indicates the document should transition to an inactive state. In a GUI, this is called closing the document and the document just disappears, sometimes after displaying a modal dialog box warning you to save it. If you didn't save it or you don't already know how files and folders work, then your content is lost. Figure 2.4 shows a better way to handle the transition in NaturalDoc.

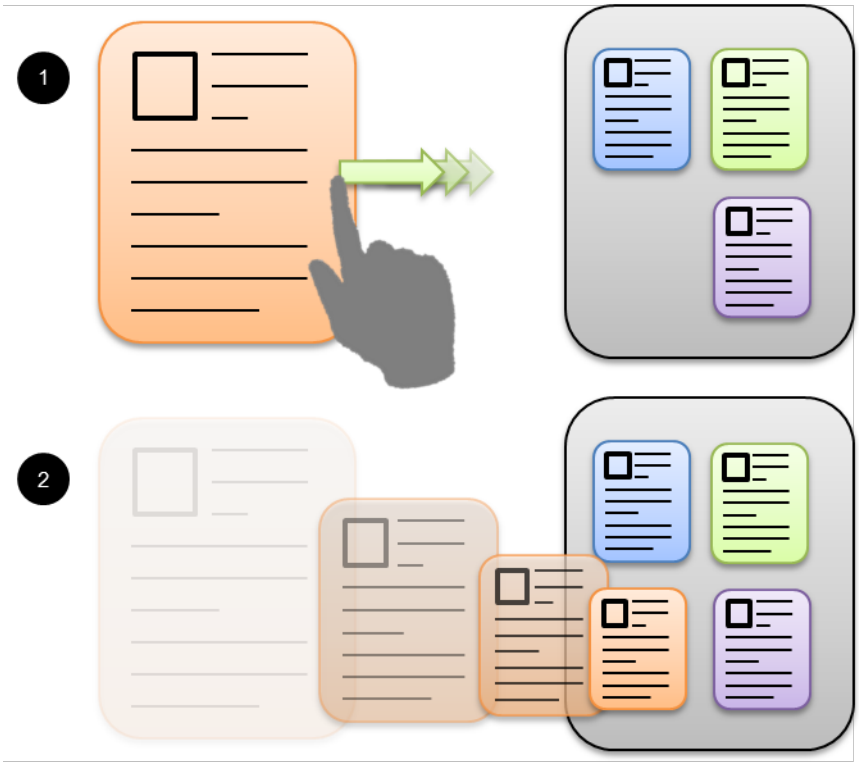


Figure 2.4 In NaturalDoc, when the user flicks an active document towards the storage container #1, the document automatically transitions to an inactive state #2. Inactive are organized in a grid in the storage container, and the transition from active to inactive is fluid.

In NaturalDoc, we can use the object metaphor and super-realism to create a much more natural experience. After the user triggers the transition to inactive, the document object accelerates towards an open spot in the storage object's container. At the same time, the size of the document object animates so that when the document reaches its target, it is the right size to visually fit in the container. This whole sequence completes in only half a second.

None of these animations have a real-world equivalent. (When you finish a real-world book and close it, it doesn't magically fly back to its spot on the bookshelf.) The use of physicality is easily understood, though, even when they are super-real. This particular implementation of physical behaviors has a bonus feature: the animation helps the user spatially understand what happened to the document. The user will understand the document is now inside of the storage object and can go there to find it again.

This example makes use of object permanence, physicality, and also demonstrates the last aspect of the object metaphor that I want to discuss: fluid transitions.

2.2.3 Fluid transitions

If you embrace the object metaphor and create permanent, physical objects, you will also need to create fluid transitions. A fluid transition is an aesthetic, continuous animation that helps the user to understand how and why an object is changing states. If you have implemented any physical attributes, you should use them in the transition as well. The previous NaturalDoc example animated the acceleration of the position and size.

Real-world objects have different states depending upon how we are using them at the time. For example, a bath towel could be described as having three states: folded in the linen closet, hanging on a towel bar, sitting in a clothes hamper. Each of these states has a completely different physical configuration and there is a transition to and from each state.

The transition from one state to the next does not happen instantaneously. There is a transition between each state, and in this case the transition occurs entirely through physical manipulation. Someone has to grab the towel from the closet and put it one the bar, use it to dry off and put it back on the bar to dry, move it in the hamper, then wash and fold it and return it to the closet. The towel fluidly transitions between each state.

There are three main benefits of using fluid transitions between states: user orientation, discovery, and aesthetic appeal. Let's briefly go through these.

USER ORIENTATION

Fluid transitions help the user maintain their orientation in the interface. This means that the user knows the current state of the objects in the interface and understands how to transition from the current state to another state. The user manually causes a transition, like pinching a NaturalDoc document down and putting it away, or the user or external event could trigger an automatic transition, such as the user gesturing for a NaturalDoc document to put itself away. In both case the fluid transition and animation keeps the user oriented about the state of the interface and suggests how to reverse an action if necessary.

> **NOTE**
>
> The words state, configuration, and mode are generally interchangeable when discussing a transition, although each one has a different connotation and may be appropriate in different situations. "State" would be appropriate for a major transition in behavior. "Configuration" would be appropriate when an object changes its shape beyond simply scaling the size. "Mode" may be appropriate when a group of interface elements are disabled or disappear and are replaced with a different group of interface elements. "Mode" has a strong link to GUI, though, and it may be better to avoid.

In NaturalDoc, the document object has two states: active and inactive. An object can have more than just active and inactive, though. The object might need to change the level of detail to present to the user. As the user shrinks an object it could hide less important information, then show it again if the user enlarges the object again. Depending upon the complexity of the content that an object represents, it might make sense to hide certain information or interface elements until the user needs them. In both of these cases, the transition from one configuration to another should be fluid. This can be achieved through clever application of physical attributes and simple animations of the interface elements.

DISCOVERY

Fluid transitions can also help the user discover how to interact with the interface. Consider how object appear for the first time to the user. If the user is supposed to be able to move and rotate an object, you could demonstrate this by having the object transition into existence while moving and rotating, even just subtly. This will give the user an idea of what an object can do on its own and encourages the user to try it out themselves.

With more complicated object interactions, this discovery is particularly important. Recall the SurfaceCube application discussed in section 1.4.4. The mini-cube performed an animation on its own when it first appears and repeats it every twenty seconds if the user does not touch it. This transition into existence and repeat animation helps the user discover how to interact.

AESTHETIC APPEAL

Don Norman, an accomplished and influential usability expert, designer, and author, likes to say that attractive things work better. Traditional scientific approaches to design cannot explain this connection, but there is an emotional justification. Emotion is tied in with stress, anxiety, and the fight-or-flight response. When we have a negative emotional reaction to something, it focuses our minds to help us in a possibly dangerous situation. When we are overly focused, we cannot come up with solutions that require creative thinking. On the other hand, when we have a positive emotional reaction to something, it relaxes our minds and allows us to think creatively. Creative problem solving is often required for the tasks that we perform with computer interfaces.

Fluid transitions and well-design animations can put your users into a good mood, which will help them perform tasks with your interface better and ultimately they will like your application better. Details such as easing in and out of animations, adding a little rebound to an animation when appropriate, and making sure the visual design is polished may seem trivial and sometimes silly, but they add up to a better emotional experience for your users.

We have now discussed the details of the object metaphor and how permanent, physical objects with fluid transitions make use of natural behaviors. The object metaphor represents most of the attributes of content except one: the relationship between different pieces of content. That is where the container metaphor comes in.

2.3 The container metaphor

Containers are metaphors for the relationship between content. With the container metaphor, objects are associated with a container designed to help the user understand relationships between the content that the objects represent.

To use the container metaphor, you need to know what the metaphor itself is, how containers and objects relate, and some ideas for how you can design useful and interactive containers. Let's start with table 2.4, which shows the container metaphor in detail.

Table 2.4 Containers map spatial relationships to logical relationships

Metaphor	Source domain	Target domain	Mapping	Example
Containers	Real-world containers	Content relationships	Logically grouped content is organized through spatial relationships	Map containing geotagged images

The container metaphor is based upon our early experiences with real-world containers. Those experiences form the basis of our understanding of both physical and metaphorical containers and the containment relationship, which we discussed in section 1.4.2. (It is not an accident that we discussed object permanence and the containment relationship when discussing abilities and skills in chapter 1.)

The container metaphor draws from our understanding that containers imply something about the relationship of the items they hold. This relationship could be as simple as each item is from the same class of objects (all toys go in the toy bin) or could be as complicated as representing multiple dimensions of data (DVDs sitting in a shelving unit organized by category and then name).

With a shelf of DVDs, the shelf is a container and the DVD case is an object. DVD cases contain the actual DVD itself, and sometimes they contain multiple DVDs. In this case, and object can have a container. Let's examine how containers and objects can be composed.

2.3.1 Composing containers and objects

Objects can be inside of a container, and containers can also be embedded in an object. This is natural pattern from the real-world where you can have several layers of containers. Junk food is commonly individually wrapped, and those wrappers come in a box, and those boxes at some point were in larger boxes, on a pallet, in a truck, on a highway, and so on, each layer being its own object but also serving as a container for other objects.

When we design interfaces with OCGM, it's important to keep the ideas of objects and containers separate. If an object needs to contain other objects, I like to say that an object has an embedded container, rather than an object is a container. The reason is that the since objects represent content, the object likely has other information to display in addition to the fact that it contains other objects.

Since objects and containers can be composed, you can use them to present hierarchies of data. Let's look at how we can transform hierarchical data into OCGM.

TRANSFORMING HIERARCHICAL DATA INTO OCGM

One approach to designing objects and containers would be to start with the database. Suppose we are designing an IT administration NUI with OCGM and we have a database with a simple data model, shown in figure 2.5, where the top-level entity is a rack. This rack has some data, such as name, location, etc., and it also has a list of servers associated to it. Each server has some information such as server name, IP address, and server type. If this server is a web server, it may also have some associated web applications.

Figure 2.5 A entity-relationship diagram showing the data model for an IT administration application.

We can start with this data model and design the objects and containers for this application. We can easily identify three objects: a data center object, a server object, and a web application object. We don't include a server type object because even though it is a separate entity in the database, it is just a lookup table and is only separate for database design reasons. The server type is really part of the server data, so we will present it as such.

Thinking about the server object for now, the visual of the server object needs to have some space dedicated to displaying information about the server. There also needs to be some space dedicated to displaying the container for the web applications. Here we say that the web application container is embedded within the server object. The server object may have different levels-of-detail that it displays depending upon the current mode or state of the object. The mode may change based upon the visual size as the user zooms in or out or could change based upon interaction with a control. Figure 2.6 shows an example design of the different modes of the server object.

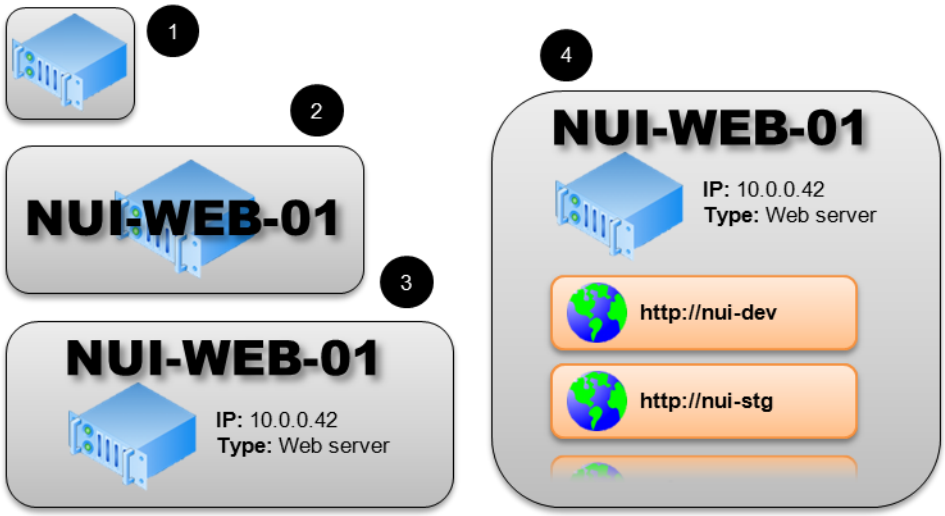


Figure 2.6 A server object always represents the same content, but depending upon the mode and how much visual space it has available it may hide some details so the user can still see the most important information in a readable way. The transitions between each mode are fluid and the content elements reflow automatically.

Cueballs in next paragraph.

If the server object is small, for example when the data center's server container is showing the all of the servers, the server object may only display an icon #1, or an icon and a name #2 if there is enough room. If the user activates or zooms in the server object, as it gets larger it can increase the level-of-detail to show more content #3 until it eventually shows all of its content, including the web application container #4.

The web application objects are presented in a scrolling vertical stack ordered alphabetically by URL. Notice that the web application container does not have any visual element, but it does fade out objects that are near the edge of the scroll viewport.

For the web applications in this data model, this organization makes sense. You have a lot of flexibility when designing containers, though. Let's talk about some of the different options and decisions that go into the design of a container.

2.3.2 Designing containers

Containers represent relationships, but how we show the relationships depends upon the concrete metaphors built on top of OCGM. A container is inherently a spatial construct, so the concrete metaphors you choose must map the different types of relationships to a spatial target domain. Table 2.5 shows some types of content relationships and how they can be presented in a container.

Table 2.5 Containers can represent different types of content relationships using a spatial metaphor. Some types of relationships can be represented using several valid metaphor mappings.

Content Relationship	Example Data	Mapping	Example Metaphor
Logical Grouping	Categories	Group to proximity	Object in the same category are positioned together
XY Spatial Data	Geotagged images	Lat/Long to map	Images shown on a map at proper latitude/longitude
Ordered Data	Task list	Order to stack position	Tasks are stacked with first one on top

Alphabetical Data	Contact list	Alpha order to stack position	Contacts are stacked with A on top
Alphabetical Data	Contact pictures	Alpha order to wrapped position	Contact images are positioned left-to-right then wrapped to the next row
Temporal Data	Network traffic	Timestamp to timeline position	Network packets are positioned on a timeline
Temporal Data	Appointments	Dates to calendar	Appointments are positioned within a calendar

The best concrete metaphors unambiguously map relationship data to a spatial presentation. In most cases we have real-world analogs we can use, such as the calendar or map. If you are able to, by all means base the design off of the real-world analog so you can reuse skills your users have learned from interacting with the real-world equivalents.

When there is no real-world equivalent, you may have to get creative. Look at the dimensionality of the relationship data. In most cases the data in the relationships are only one- or two-dimensional. That works great because you can map one or two dimensions very easily. For example, suppose you had objects that represented individual pages in a magazine. Besides the obvious and over-used linear page-flip metaphor, you could design a container to map magazine stories to horizontal positions and pages within the story to vertical positions.

Evaluating real-world interactions and metaphors

When choosing which real-life interactions and metaphors to implement, we have to consider the form factor of the device and the environment the user will be in. Then, only use interactions that add to the experience and metaphors that help the user understand the interface.

The page flip metaphor and interaction fail both of these tests in most cases. If the user needs to grab a virtual page and flip it, it takes time to display the animation and distracts from the content. A linear layout of pages and panning between them would be more efficient.

To be fair, flipping pages could be appropriate in a few narrow cases when the act of turning the page is important to the experience. One example would be if you were creating a pop-up book simulator. Another would be if you needed to simulate the experience of tracing paper or carbon paper for some reason. In most cases where you are just displaying pages of information, flipping pages is not necessary.

Unless you are creating a data visualization instead of an interactive interface, don't try to map three dimensions of relationship data into a 3-D interface. The overhead of learning and navigating the 3-D interface on a 2-D display outweighs any benefits the extra dimension would bring. Instead, represent the extra data dimension using a different container mode, or in a hierarchical fashion with another object with an embedded container. Of course, if you do happen to have a 3-D interface using a holographic display, augmented reality, or binaural audio, you might be able to get away with a 3-D interface metaphor.

Once you have an idea of what concrete metaphor you will use for a container, you have to actually design the container itself. There are three primary techniques you can use to present a container metaphor to the users: Layout, Interface elements, and Behavior.

Container Layout

The primary function of a container is to provide a context for layout, or positioning, of objects. The most basic containers may only do positioning and may not need to have any visual interface elements aside from the objects themselves. Consider the problem of group playing cards, illustrated in figure 2.7.

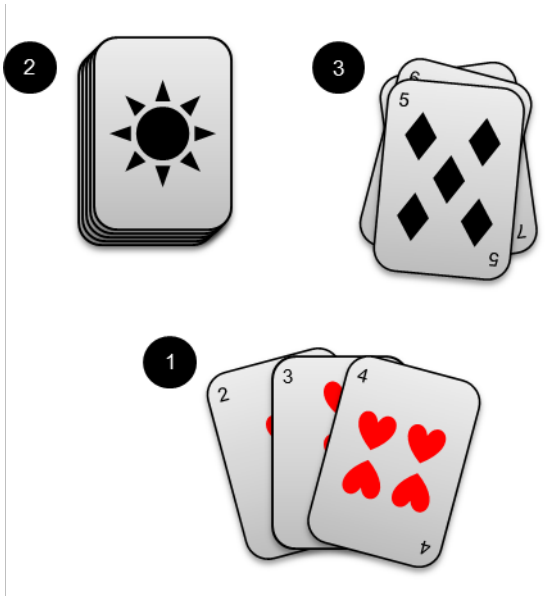


Figure 2.7 Playing cards are objects and they can be grouped into containers of a player's hand #1, the deck #2, and cards in play #3. The containers have no visual representation aside from the card objects, but the proximity of one card to another card and the layout the cards helps the user understand the relationship between the cards.

The user understands which cards are in a group and the relationship between the cards by observing their proximity and layout. In an application that uses these cards and card containers, the user could even interact with some of the containers by moving them, depending upon the needs of the application.

When I say that an object is inside of a container, I mean that in a symbolic way. As in the card example, there does not necessarily need to be a hard visual boundary separating what is "in" and what is "out". There is no bounding box around the cards in the player's hand, yet we easily understand that a particular card is "in" the hand, and the hand is a container.

The layout that you choose is based upon the concrete metaphor you are using. The layout needs to communicate something about the relationship of the objects to the user, even if there is no relationship.

The over-use of ScatterView

If you have content that has an underlying relationship, you should avoid the temptation to layout the content in a way that does not provide any clue about those relationships. A perfect example is the tendency to use ScatterViews in many early multi-touch applications for almost every type of content.

The ScatterView is useful when the content is only loosely related and the user needs to be able to sort through objects in a free-form fashion. The over-used example is displaying unrelated pictures to the user to play with using pan, rotate, and scale manipulations.

In a real application for multiple users, ScatterView could be a good choice if you have a couple top-level objects. For example, consider a retail application with a few dozen merchandise objects. A naïve design would be to throw all of the merchandise objects into a single ScatterView. This would overwhelm the user and give them no way to understand the merchandise and the relationships between types of products.

Instead, keep a top-level ScatterView but only start with a shopping cart object and a catalog object. Each of those objects have embedded containers that present merchandise products in an ordered way. The users can manipulate the cart and catalog objects and drag merchandise from the catalog to ScatterView or to the cart. This design would have a much better user experience.

When content does have an inherent relationship, it is important to present it to the user in a container that illustrates the relationship, rather than taking the easy way out and using a ScatterView.

If you want to show only ten or twenty objects, a compromise between meaningful layout and free-form manipulation could be create an "ordered" ScatterView. In this case objects would be arranged according to their relationships, but the user could pan, rotate, and scale an object. When the user is done with an object, make sure it gets back to its original position to maintain some sense of order. Depending upon the application, the object could be sent "home" when the user puts it approximately in position, when the user stops interacting with an object, or when the user performs a gesture or presses a button on the object.

Sometimes content has several types of relationships with different metaphors and the container needs to change modes to display each one. For example, you could have a task list displayed sequentially or displayed by task address on a map. In keeping with the object metaphor, the transition between these modes needs to be fluid and the tasks should animate from the screen position in the sequential list to the appropriate screen position on the map.

You would also want to account for the case where a task does not have an address by providing a holding location outside the map. The user may want to drag those unmapped tasks onto the map to set their address.

In order for the user to trigger this mode change, there will have to be some type of interface element, and possibly a gesture as well. This brings us to the discussion of interface elements on containers.

CONTAINER INTERFACE ELEMENTS

Objects may have their own interface elements, but some metaphors are best presented by adding interface elements to the container as well. These elements should be specific to the container, rather than a single object. For example, if we discuss network traffic on a timeline, the container could just position network packet objects according to the timestamp. It would help the user understand the metaphor and the information encoded into the object placement, though, if we added a timeline visual with scale labels, as in Figure 2.8.

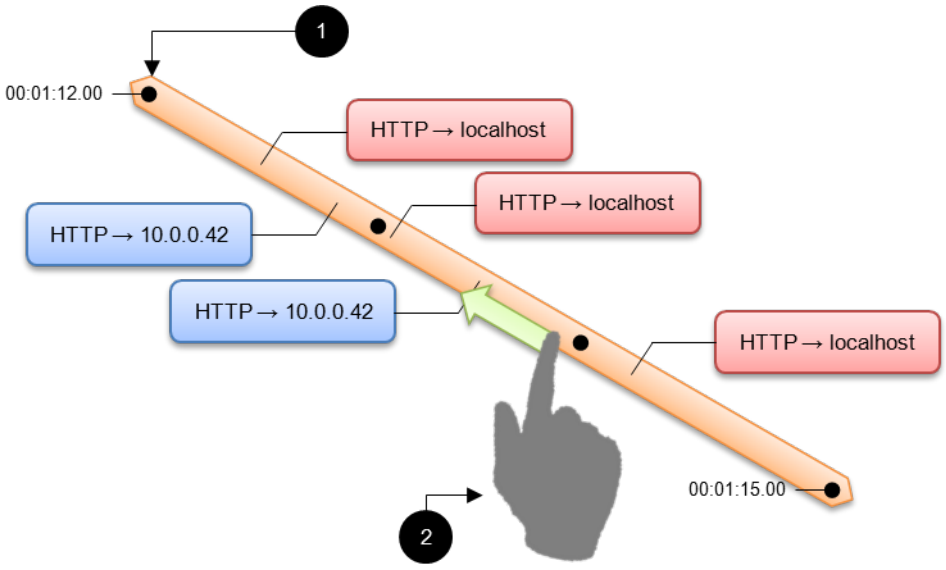


Figure 2.8 A container can position network packet objects according to their timestamps, but adding a interface element, in this case a timeline and scale labels #1, helps the user understand what the positions mean. Packet objects representing outgoing data are shown on the left in blue and packet objects representing incoming data are shown on the right in red. The timeline can be panned #2 to move the visible time window. The layout is angled so the objects can display more data without overlapping. The timeline was rotated, rather than the objects, to make it easier for users to read the information without turning their heads.

Adding a few interface elements also provides additional affordance for interacting with the container. Interactions with the container should be specific to the relationships. With the network traffic example, interacting with a packet object may let you see more detail on that object. Users can also interact with the container because the timeline provides affordance for changing how the container lays out elements. We can use a pan manipulation on the timeline to change the time window, or we could also use a pinch manipulation, as in figure 2.9, to change the amount of time shown. Just like the server objects, as more or less space is available, the packet objects change modes and fluidly adjust their appearance.

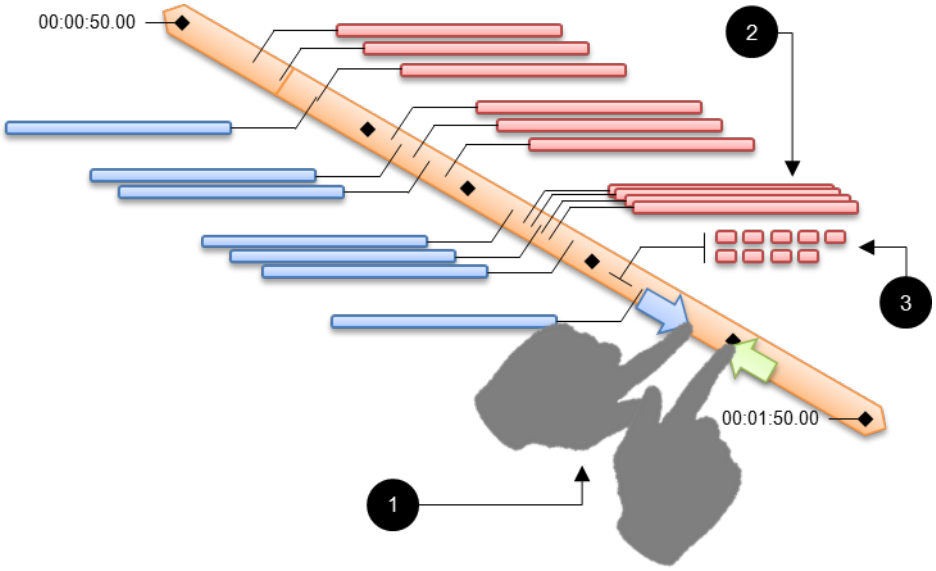


Figure 2.9 Pinching #1 the timeline scales the time window accordingly. Here we have changed from a three second window in figure 2.8 to a sixty second window. At this scale the user would primarily be looking for patterns of packet flow, rather than details of individual packets. Notice that the packet objects have changed to a minimal size appropriate for the current scale. In some cases objects may be positioned so close they overlap #2. To prevent the information from getting too busy, the container can cluster objects #3 by changing them to a smaller mode and adjusting the layout. This maintains the object metaphor and allows the users to still see how many packets were transferred at a glance.

Interface elements should only be added when they have a specific purpose. When you are deciding whether or not a particular element should be included in the container, use the questions in table 2.6 to guide you.

Table 2.6 To justify a container interface element, you should be able to answer "yes" to at least one of the following questions about the proposed element.

Question	Justified example
Does the element communicate information?	Category label
	Timeline tick marks
Is the element required for understanding or interpretation of the metaphor?	Map background
	Calendar grid and text
Does the element provide affordance for interacting with the container?	"Handle" on a ScatterView item
Is the element necessary for controlling the container features?	Control to switch container modes

If the element is a control, is it necessary instead or in addition to a direct manipulation?	Toggle between manipulation modes Toggle between layouts

I want to elaborate on the last question. If the container is a map, pan and zoom controls may not be justified because direct manipulation of the map would be preferred. Pan and zoom controls can actually prevent users from discovering the direct manipulation feature. If interaction design or user research indicates a control adds to the experience, or if there is no intuitive manipulation, then a control can be justified.

Designing a map container

To illustrate the decisions that go into the design of a container, let's examine a detailed example. The challenge is to design a map container that has these features:

- Primary interaction with the map is standard pan, zoom, and rotate manipulations
- Allow the user to switch between pan/zoom/rotate and 3-D perspective tilt modes

Designing the pan, zoom, and rotate manipulations is fairly easy. On the other hand, there is no easy way to change from a pan/zoom/rotate mode to a 3-D perspective tilt mode without using manipulations that are confusingly similar to primary map manipulations.

One solution to this problem might be to add a gesture. On a multi-touch system, you could design it so when the user places more than a certain number of fingers on the map, it behaves differently. This may not be the best solution, though, because in a multi-user scenario you may get accidental activations of this gesture, and differentiating the number of fingers is not a natural thing to learn. In the real world, objects behave the same regardless of the number of fingers we use.

Depending upon the capabilities of your touch device, you might be able to differentiate between large blobs or other objects and finger tips. On Microsoft Surface, for example, you could place one palm down to temporarily change modes and manipulate the map with a second hand. This is also not necessarily natural, for the same reason as counting fingers is not natural. Alternatively, you have Surface recognize an object using a byte tag to cause a change. This would be natural because the physical object can be used as a tool.

Regardless of any gesture for changing manipulation modes, you may choose to use a control on the container as well. This improves the novice user experience because a control is easily discovered. Over time the novices can learn the gesture and perform the same mode change in a quicker way.

You may think that the questions in table 2.5 will lead to a minimalistic interface, and you would be correct. Since NUIs focus on direct interaction with content, we do want to minimize the non-content interface elements. We want to start with a default of just content, and then justify everything we add to it.

This does not necessarily mean that a container has to be boring. You can still design the interface elements that you do have to be interesting. As an example, I designed a Surface application that has a historical mapping theme. I needed a scrolling list of maps that the user so the user could drag a map from the list onto the main workspace. This list also had to be a ScatterView so users could move it around. The final design of the historical map themed scroll list is shown in figure 2.10.

Figure 2.10 The final design of a historical map themed scroll list. The top and bottom scrolls served as handles so the user could grab the list and move or rotate it as a ScatterViewItem. The center area scrolled up and down to reveal additional maps. If the user touched a map and dragged left or right, a drag-drop of the appropriate map would start. When users are done working with a map they can drop the map over the scroll list and it will animate back to its place in the list.

This scroll list itself is a container and it contains all of the map images in a sequential list. The metaphor for this container is a literal scroll, and the interaction is designed to look like the map images roll out and in the scroll ends. Even though the theme is obviously presented, each interface element is justified in use according to the questions from table 2.5. The justifications and the theme design are explained in table 2.7.

Table 2.7 The historical map themed scroll list contains several different interface elements. Each of them is justified first and then designed to fit the theme.

Interface element	Justification	Theme design
Map image corners	Corners provide a picture album affordance to indicate the images can be removed	Corners are designed to look like an old-style picture album.
Map image labels	Labels provide additional information that helps the user select a map	Font face chosen to look hand-written in the proper period
Scroll list background	Background provides additional affordance for scrolling and helps the understanding of the literal scroll metaphor	Background image chosen to look like parchment appropriate for the period
Scroll list ends	Scroll ends complete the literal scroll metaphor and provide an affordance for a non-scrolling area for moving the list around	Ends look like rolled up parchment

You can see from this example that the scroll list is minimalistic, but it does not have to be boring. Each of the included interface elements served a particular purpose, and if any were removed interaction with the container would seem incomplete. It was not necessary to include a scroll bar or any other controls. Nothing needed to be added or taken away.

An important part of the design of this scroll list was the interaction with the maps and how you could drag them out or drag them back in. This leads us to the final aspect of container design: the behavior.

Container Behavior

When you design a container, you also have to consider what behaviors the container's metaphor gives to the objects. Table 2.8 lists some ideas for metaphors and behaviors.

Table 2.8 Containers affect the object's behavior depending upon the metaphor used. These behaviors are some ideas of how manipulating containers can add to the experience.

Metaphor	Action	Result
Any	Pinch on object	Increase object scale and level-of-detail
Any (Magnetic)	Manipulation that changes object or container modes	Objects and containers animate to the nearest mode once the manipulation ends. Objects cannot be left between modes.
Proximity	Pinch anywhere	Scales all objects up or down
Proximity	Pinch past a scale threshold	Objects animate to a new layout appropriate for scale
Free-form	Manipulate object	Pan, rotate, scale manipulations affect individual objects
Vertical stack (scrolling)*	Vertical scroll	Drag the stack vertically to scroll the viewport into the stack
Vertical stack (fixed)*	Vertical rearrange order	Drag an object to rearrange the order**
Vertical stack*	Horizontal drag on object	Drag an object out of the stack
Vertical stack*	Vertical pinch	Rescale vertical axis, increase size and level-of-detail that objects display

Any stack	Object dropped over stack	Insert an object into the stack
Zooming canvas***	Manipulate anywhere	Pan, rotate, scale manipulations affect the canvas as well as any contained objects
Zooming canvas	Manipulate background	Pan, rotate, scale manipulations affect the canvas as well as any contained objects
Zooming canvas	Drag object out of canvas	Removes object from canvas
Zooming canvas	Manipulate object into/within canvas	Add/move object in canvas, sets new position/orientation/scale

* Also applies with vertical and horizontal switched
** Similar to dragging icons to rearrange the Windows 7 taskbar
*** Maps also use a zooming canvas metaphor

Note that not all of the behaviors in table 2.8 can be used at once. Even within a single metaphor, for example the zooming canvas, the "manipulate anywhere" action and the "manipulate object" action conflicts because "manipulate anywhere" overrides manipulating a single object. You could use "manipulate background" and "manipulate object" at the same time. If you wanted to implement both behaviors, you could implement a way to change between one mode and the other.

To illustrate how the "pinch past a scale threshold" behavior might be combined with a "magnetic" interaction, consider a container that holds pictures, shown in figure 2.11.

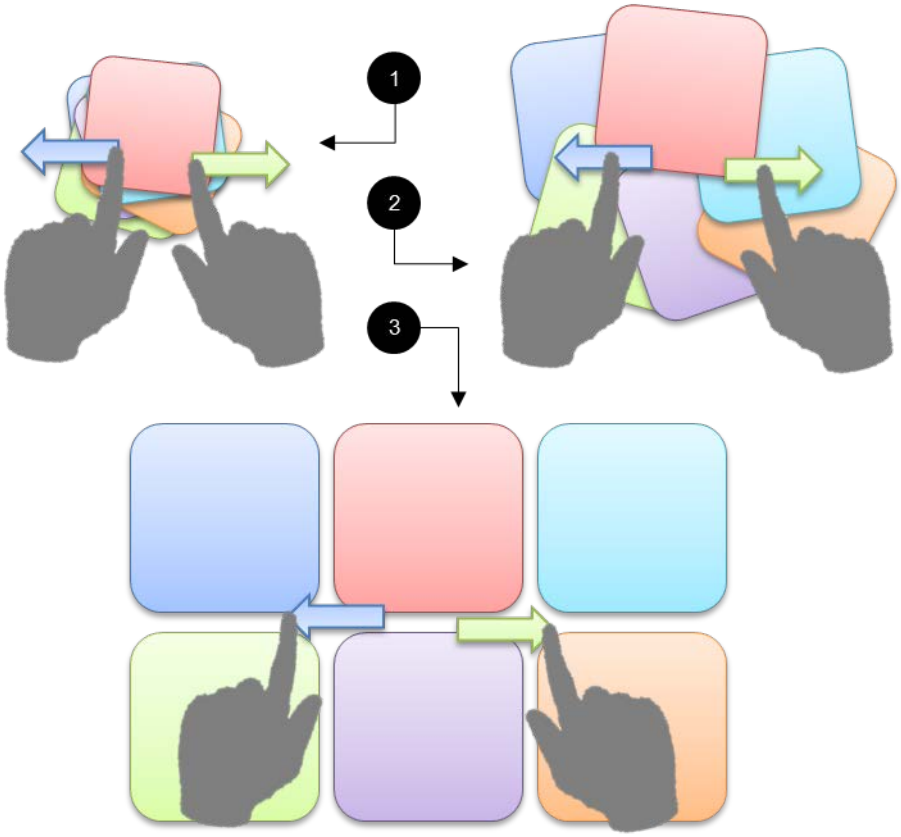


Figure 2.11 Contains can just be about layout and behavior. This picture container starts out by laying out pictures is a pile #1, each with slightly randomized rotations. As the user pinches out, the pictures each grow in size and move outward from the center #2, allowing the user to peek at the pictures that are lower in the pile. As the pinch crosses a threshold, the container starts moving the pictures towards positions in an ordered grid #3. This interaction is magnetic because at if

the fingers are removed in the middle of the transition, the container will use an eased animation to finish the transition. If there was no inertia when the fingers were removed, then the container animates to the closest mode. If there is inertia, then the container continues animating towards the appropriate mode.

In the default state, the pictures are displayed as a slightly disheveled pile. If the user performs a pinch out manipulation on the pile #2, the pictures scale up and move outward from the center of the pile. So far it is just a basic "pinch anywhere" behavior. After the cumulative manipulation reaches a certain threshold, the layout of the container starts to transition from disorganized pile to ordered grid #3. If the fingers are removed at this point the container uses a smooth eased animation to move the pictures to their final positions. Pinching back in past the threshold changes the layout and the pictures ease back into the pile.

In an actual implementation of this behavior, you would need to implement a dead zone in the threshold to account for noise in the input data. If the current state is the pile, the layout would only change to grid once you scale above threshold + dead zone. Likewise if the current state is the grid it would only change to pile once you scale below threshold - dead zone.

We've covered a lot of different ideas for implementing objects and containers. I hope that you have seen through the examples I've chosen how OCGM and NUIs in general can be applied to many different types of interfaces with diverse applications.

2.4 Summary

This chapter gave you a primer on OCGM and metaphors and focused on how to approach the object and container metaphors. Objects and containers are the foundations of any NUI interface element. Compared to the beginning of the chapter, you should have a fairly concrete idea of how to design an interface that implements some of the NUI concepts from chapter 1.

To get a complete picture of how to apply the rest of the NUI concepts, we need to talk more about how we interact with the interface as well. We have discussed a little about manipulations and gestures in this chapter, but in chapter 3 we will get into the details. We will discuss the differences between manipulations and gestures, when it is appropriate to use each, and see examples of how to create interactions with specific manipulations or gestures in mind.

Chapter 3 will round out part 1 and you'll be armed with a significant amount of knowledge that you can use in the rest of the book as we learn about the touch APIs and try out creating real NUIs, not just illustrations and concepts.

4

Your first multi-touch application

I know what you must be feeling right now. Multi-touch development is a new present, and you can't wait to rip the wrapping paper off and start playing around. Don't worry; I'm not going to get in your way. This entire chapter is all about getting you up and running with your first multi-touch application. Granted, this first application is not going to be the most amazing NUI, but my hope is that this will get you started with minimal effort.

In a multi-touch natural user interface, you are creating the interface magic at the top level of several layers of abstraction. You write the application on a platform with a multi-touch API. For us, the platform is Windows 7 and the API is the WPF 4 Touch API. The platform in turn depends upon a wide range of hardware to send it touch data. To fully understand the behavior of your applications, you must understand each of these layers.

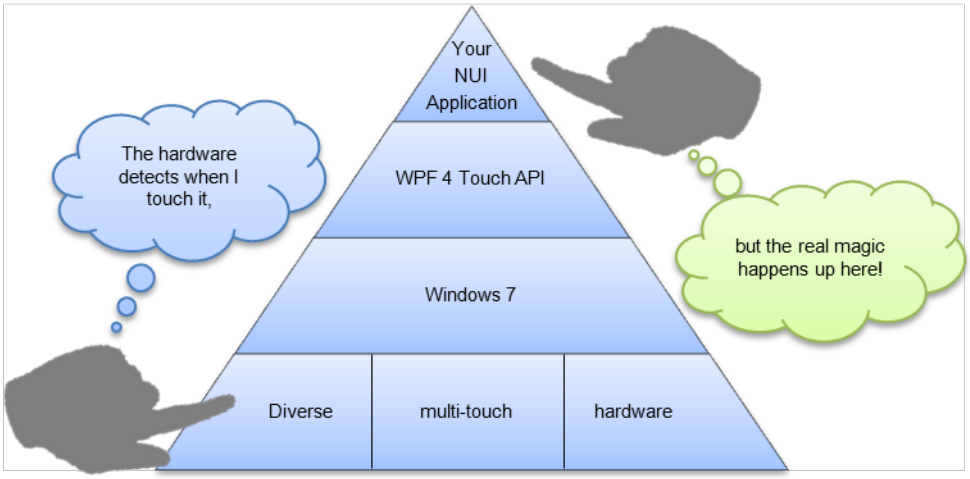


Figure 4.1 There are several layers of abstraction involved in getting the touch data from the fingers to your application. Windows 7 provides a standard platform for many different multi-touch devices, and WPF 4 provides a great API for creating multi-touch applications. All you need to do is create a great natural user interface!

We are going to discuss the platform and the API, but first let's start at the fingertips out and talk about the touch hardware.

4.1 Getting to know touch hardware

I expect that if you are reading this, then you either already have a computer with a multi-touch display, or you are seriously considering buying one in the near future. In both cases, you should understand there are several different technologies that can enable multi-touch.

Each type of multi-touch technology behaves slightly differently. To understand the individual behaviors, you need understand the basics about how the technologies sense touch. Once you understand the technology, you see how some technologies have quirks and edge cases in how they recognize and report touches. When you plan your application, you will definitely need to know how many simultaneous touches the hardware will report. We are going to cover each of these topics, starting with how the hardware detects touches.

Since new hardware and technologies are coming out all the time, it wouldn't help to evaluate specific products here. Instead, let's talk about the general classes of multi-touch display technology and their characteristics.

Touch Terminology

There are several words commonly used when discussing touch hardware. Let's review the definitions.

surface - Any two dimensional area that detects touches. Not to be confused with Microsoft Surface, a specific touch table product

touch screen or touch display - A computer display that has an integrated touch surface. Touches should coincide with the displayed element it is interacting with.

touch tablet or touch pad - A non-display touch surface. Graphics tablets traditionally use stylus input and are targeted at graphic designers. Not to be confused with Tablet, a laptop form factor.

digitizer - Hardware that actually detects and tracks the touch contacts. Optical multi-touch techniques use a camera instead of a digitizer.

touch - The point at which a finger comes in contact with a touch surface. Can also refer to the data generated by a touch.

contact - A more generic term for touch that includes finger touches as well as other detectable forms, such as stylus or blob

blob - A general term for an unidentified touch shape. Blobs are often represented using ellipses calculated from the blob's bounding box, but the actual touch may be a more complex shape, such as a handprint.

multi-touch - One of several technologies that are capable of detecting and tracking two or more simultaneous touches

massive multi-touch – Multi-touch technology where the maximum simultaneous touches is not limited by hardware. Generally, devices capable of massive multi-touch can report at least ten and up to fifty or more simultaneous touches. Also referred to as true multi-touch.

4.1.1 Touch Hardware Characteristics

The minimum requirement for a touch screen to be considered multi-touch is the ability to track and report on two or more physical touch contacts simultaneously. There are a variety of techniques that allow this, including capacitive, resistive, optical, plus more exotic methods such as acoustic. Let's review the basic concepts behind these technologies.

CAPACITIVE

Capacitive touchscreens take advantage of the electrical properties of human skin. The digitizer can detect an electrostatic charge when a finger is in contact with the device, as shown in figure 4.2.

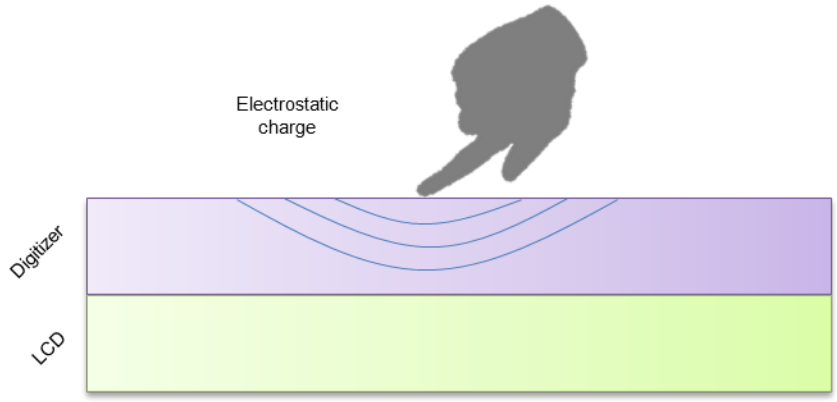


Figure 4.2 Capacitive screens detect touches by using a digitizer layer that is sensitive to the electrostatic charge of fingers. This digitizer is transparent and is layers in front of a standard LCD screen. As a finger makes contact with the digitizer, the electrostatic charge is registered and the device reports the touch to the operating system. This diagram is not to scale.

Capacitive screens require no contact pressure and can also be integrated with an electromagnetic digitizer that can detect stylus input. Capacitive touchscreens are commonly used in consumer electronics such as cell phones and multi-touch tablets. One disadvantage of capacitive is that it requires direct contact with a finger, so you cannot operate one while wearing a glove.

RESISTIVE

Resistive touchscreens have two conductive layers separated by insulating layer or gap. When something presses on the screen, the layers are connected electrically and the screen can detect the position of the contact. The screen can then tell the position of the touch. This is illustrated in figure 4.3.

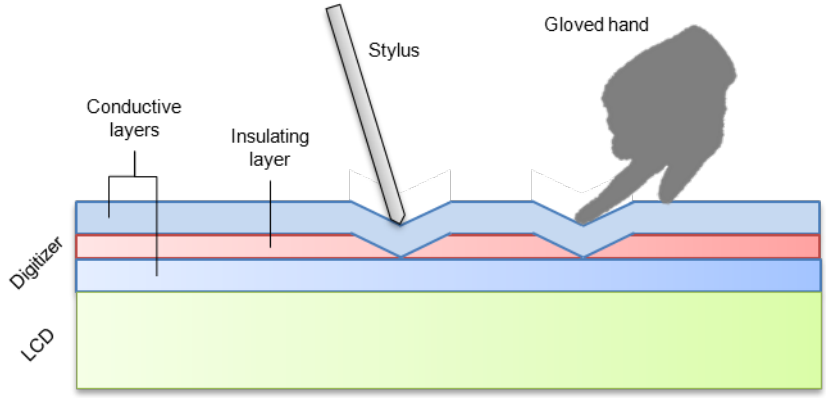


Figure 4.3 A stylus or gloved finger can work with a digital matrix resistive touchscreen. The pressure from the contact deforms the conductive layers enough to cause an electrical connection. This connection allows the digitizer to detect the position of the contact. This diagram is not to scale.

When considering resistive touchscreens, you must determine whether it is using an older, analog technology such as 4- or 5-wire or the newer digital matrix technology. The analog variety is of lower quality, requires more pressure, and is usually not capable of multi-touch. The newer digital matrix resistive is better quality, requires almost no pressure, and is capable of multi-touch.

One advantage that resistive has over capacitive is that it responds to any touch. You could use a gloved hand or stylus just the same as a finger. A disadvantage of resistive is that it requires a minimum

level of pressure to register the touch; although, recent advances have lowered that minimum to very small levels.

OPTICAL

There are a wide variety of optical sensing techniques, most using some variation of infrared (IR) light, a camera and image processing techniques. In each case, touches are detected when IR light is reflected or disturbed in such a way that a camera can detect it. A few common techniques are described in table 4.1.

Table 4.1 There are several different optical techniques that can be used to detect multi-touch.

Technique	Description
Frustrated Total Internal Reflection	IR light is shined into the edge of a glass layer where it reflects internally between the surfaces. Touching the glass interrupts the reflection at the contact point, which can be detected by an IR camera below the surface. Images are projected onto a projection surface layer.
Diffused Illumination	IR light is shined onto a diffuse material (the diffuser) below a glass layer. An IR camera below the surface can detect touches by watching for changes in the amount of reflected light where the glass is touched. Images are projected onto the diffuser. This technique is shown in figure 4.4.
Laser Light Plane	Similar to diffused illumination except instead of an IR light illuminating the diffuser, an IR laser is used to create a plane of IR light just above the touch surface. When a finger interrupts the plane, IR light is reflected onto a diffuser where the IR camera can detect it. Images are projected onto the diffuser.
IR Overlay	Similar to laser light plane except instead of requiring a project, it can be overlaid on a traditional CRT or LCD. This technique, shown in figure 4.5, uses two or more IR cameras located at the corners watching for reflected IR light.

Optical techniques typically are capable of massive multi-touch. Microsoft Surface uses a variation of the Diffused Illumination technique, illustrated in figure 4.4.

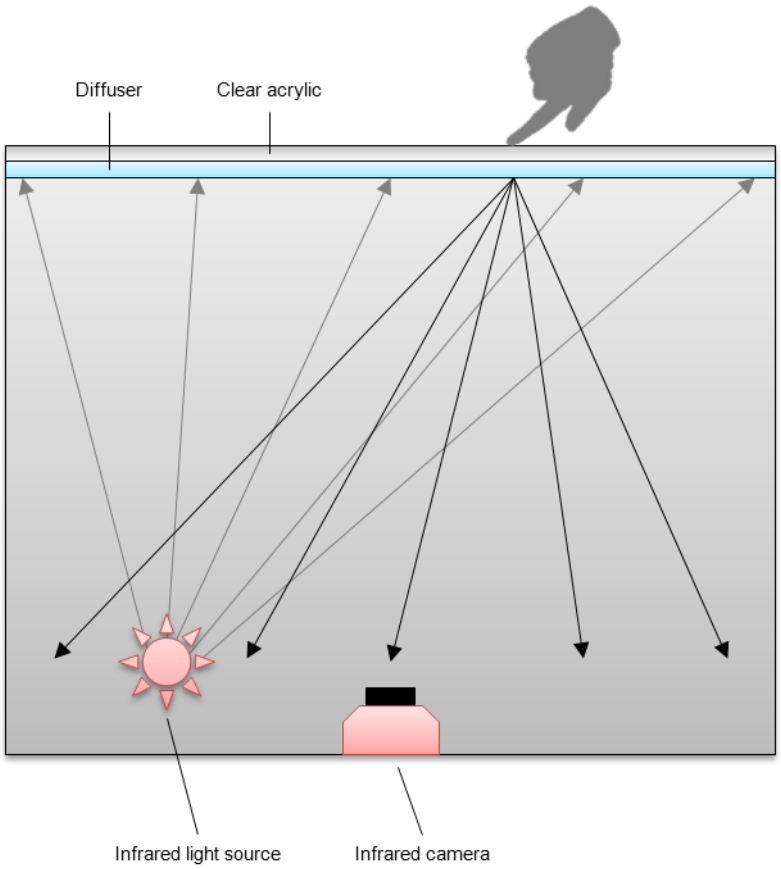


Figure 4.4 A schematic view of the rear diffused illumination technique. An infrared light source evenly illuminates a diffuser. When a finger touches the acrylic surface, the infrared camera can detect the increase in reflected of infrared light.

Optical methods are popular with do-it-yourself multi-touch enthusiasts because the materials are readily available and techniques are simple enough for a hobby projects. A large community has grown around these techniques at www.nuigroup.com.

EXOTIC

We've looked at the most common multi-touch hardware techniques, but there are also more rare and exotic ways to detect touches on a surface. Researchers have turned any surface, such as a wall or table, into a multi-touch surface by using multiple microphones and advanced sound processing algorithms. There are also ways to detect touches through mechanical means, such as force sensors on a surface's mount points, but that may not detect multiple simultaneous touches. Most exotic techniques are only used in research and not ready for mass production.

Each of these techniques has different behaviors and in certain circumstances the expected behavior breaks down. Let's take a look at some of the unexpected things that can happen with certain touch technologies.

4.1.2 Quirks and Edge Cases

It is important to know the normal capabilities of hardware, but just as important to know what happens in non-optimal conditions.

For example, figure 4.5 shows how a touch could be reported before the finger makes contact with the surface with certain optical techniques.

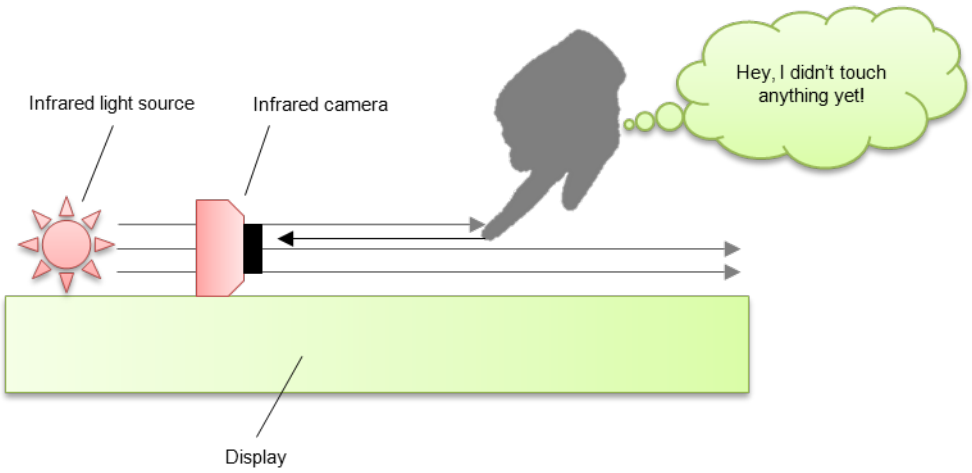


Figure 4.5 Laser light plane and IR overlay (shown here) techniques are prone to false positives because fingers can be detected before they touch the screen. This can frustrate the user if they hold their finger over the screen, thinking about a choice, but the screen reacts as if they already touched it. This can also occur if a something besides the finger, such as a wrist or sleeve, breaks the light plane.

If a touchscreen will be used for writing with a stylus, palm rejection is important because people tend to rest their hands on the writing surface. Palm rejection detects and ignores accidental touch contacts from the palm of your hand. Capacitive touchscreens combined with a stylus digitizer have an advantage over resistive because the two different digitizer make it easy to tell the difference between fingers and a stylus. Resistive touchscreens must rely on filtering based upon the size of the contact.

Optical touchscreens that use infrared can be sensitive to the environment. If there is too much ambient infrared light, such as outdoors, the device may be too sensitive and register false touches.

Finally, the behavior of the touchscreen when too many touches are present can also be important. If a touchscreen is designed for two touches and the user accidentally touches a third finger, which two of the three contact points will be reported to the software? Some hardware will alternate between two of the three, and some will average the positions.

Keep these factors in mind if you are lucky enough to be able to choose the hardware you or your users will use. If you are creating an application that will be distributed to the general public, you may not have that choice. Some devices are more reliable and capable than others. For example, devices that use optical techniques like the one in figure 4.4 are often the most finicky. Make sure to test your application with the range of hardware your users will likely be using.

How users will use your application will depend upon the form factor of the hardware.

4.1.3 Form factor

The term form factor refers to several physical attributes of the hardware. Specifying the form factor requires knowing at least the size, shape, orientation, and mobility. The form factor will determine the scope of how users can use your application. Certain form factors, such as tablets, are typically limited to one user at a time, while larger form factors such as horizontal table touchscreens enable multiple people to use your application at the same time. Table 4.1 lists several form factors and typical use scenarios.

It is important to take into account the target form factor when creating your application.

Table 4.1 describes several types of touchscreen form factors.

Table 4.1 Multi-touch screens come in many different form factors. The form factor determines what type of interaction is practical.

Form factor	Scenario
Phone	Single user, used alone or socially, easy to pass from person to person
Slate	Single user, used alone or socially, easy to pass from person to person
Tablet	Single user used alone or occasionally with others watching
Vertical screen	Single or multiple users, users approach side-by-side, individual or collaborative tasks
Horizontal screen	Single or multiple users, users can approach from any direction but often pair on one side, individual or collaborative tasks

When starting a new project, be sure to note the type of collaboration desired and ensure the appropriate hardware is being used.

Independent of the form factor, the number of touches the hardware supports also an important consideration. The number of touches will affect the type of actions the application can support, so let's talk about the how the number of touch affects the interactions.

4.1.4 Number of Touches

In general, the specific type of touchscreen hardware will not make a big difference to you as you create multi-touch applications. The only thing that can be a huge factor is how many simultaneous touches are possible.

The number of touches places a limit on the complexity of gestures and manipulations as well as the number of people who can interact at once. Think about how you interact with the real world using your hands and consider how you use hand gestures when communicating with other people. Now imagine that instead of five fingers on each hand, you only had a single finger on each hand. Only having two touches limits what you can do, but with some creativity you can still accomplish many tasks. Now suppose you only have one finger total. Your possible interactions suddenly become extremely limited. You would need to do things in a very careful way in order to maintain any level of productivity.

Transitioning back to interfaces, the GUI world only uses a single point and like the example above you are limited in what you can do. Many of the stale GUI interaction patterns are only necessary to help you be productive. Two-touch devices have many more possible interactions than single-touch devices or the mouse, but there is still a limit. Once you grow to massive multi-touch, the number of interactions becomes unlimited, and we are back to being able to make full use of our ten fingers.

Let's explore this a little more. You can group touch hardware into three general categories: single touch, limited multi-touch, and massive multi-touch.

SINGLE TOUCH

With a single touch, you are extremely limited in the kind of gestures and manipulations you can perform. Typically, single touch touchscreens are used as a replacement for the mouse, as in kiosks, and would not be suitable for many of the advanced interface concepts that are discussed in this book.

On the other hand, there are still many interesting interfaces you can create with a single touch. If you consider the iPhone, even though it has multi-touch, the majority of applications only take advantage of a single touch. As long as a single-touch application is designed around natural human behaviors, it can still be a natural user interface.

LIMITED MULTI-TOUCH

Limited multi-touch devices are generally limited by the hardware technique to a low fixed number of simultaneous touches, often only two to four. Many cell phones and multi-touch tablets use capacitive touchscreens and are only capable of two to four touches.

Compared to single touch, two touches allows for a whole new class of gestures and manipulations compared to single touch. Two touches is the minimum requirement for multi-touch interactions. Two to

four touches is plenty for many single-user applications. Novice users tend to only use a single finger per hand even if the screen can support more.

MASSIVE MULTI-TOUCH

Massive multi-touch devices are based upon scalable hardware techniques that do not limit the number of touches. The effective maximum is only limited by processing power, and usually in the range of ten to fifty or more touches.

The richness and number of possible interactions increases significantly once you get past four touches. Massive multi-touch also enables more social scenarios, such as two people interacting with the same device at the same time.

Most multi-touch technologies have the ability to report a unique identifier, the contact position, and the contact size for each touch point. Some technologies can report additional data, such as pressure. Pressure can be calculated from the contact size, since the harder you push your finger the more surface area it has. Some optical technologies, such as Microsoft Surface, can also report on orientation and can use additional visual processing to detect markers such as byte tags or identity tags, shown in figure 4.6, which allows interaction with real-life objects.

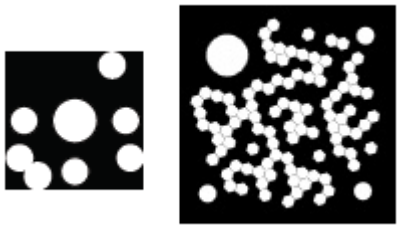


Figure 4.6 Microsoft Surface uses an optical multi-touch detection technique and can use the visual data to detect these markers if they are placed on the touch surface. On the left is a byte tag, named because it contains one byte of information. On the right is an identity tag, which contains 128 bits of information. These tags can used to enable a seamless experience between the touch surface and tagged real-life objects.

Multi-user social interaction and multi-touch gestures and manipulations all depend upon the capabilities of the touch device, but a device by itself is useless. The applications that you are going to build that make the hardware come alive. You want to stay high-level and focus on implementing great natural user interfaces, but you don't want to be mucking around with device drives and interpreting raw hardware data.

That is where the software platform comes in, as a layer of abstraction between the hardware and the applications. Luckily, there exists a great platform for multi-touch development. You may have heard of it.

4.2 Introducing the Windows 7 multi-touch platform

While previous versions of Windows supported single-touch input natively and multi-touch could be added in the application layer, Windows 7 is the first desktop operating system to natively support multi-touch displays and treat touch as a first-class input alongside the mouse.

> **NOTE**
>
> In documentation, Windows 7 and the various multi-touch technologies are generally referred to as touch, for example the WPF 4 Touch API. The use of the word "touch" is meant to contrast other types of input like mouse or stylus. The fact that "multi" is not explicitly stated in the documentation should not be taken to mean multi-touch is not enabled.

Let's take a look at the different ways Windows 7 enables multi-touch development.

4.2.1 Windows 7 Touch programming options

Microsoft designed Windows 7 so that all applications are usable with touch at a basic level. This basic level is not very natural, though, because most applications still use the WIMP metaphor and have GUI interaction patterns that are hard to use with touch. Thankfully, if you want to create a better touch experience, there are options. To help explain the different options available to developers, Microsoft has coined the "Good-Better-Best" software stack for touch development.

GOOD

Applications fall into the "Good" category if they are totally touch-unaware and the application developers made no effort to use any touch APIs. If a user touches an application and the application does not handle the touch event, Windows 7 will promote the touch event to a mouse event, and the application should respond to it. Windows 7 will also detect certain gestures, such as zoom or scroll, and send the equivalent keyboard and mouse events.

These considerations allow most areas of an application to work, but not all. For example, horizontal scrolling will not respond to touch without specifically programming it to. Only the first contact will be promoted, so this means these applications will only be single touch.

"Good" applications will almost always be GUIs. Even though users can use these applications with touch, it is really a poor experience. Complicating this is the fact that many interface elements, such as buttons, are designed with the mouse in mind and are too small for fingers to easily target.

BETTER

An application falls in the "Better" category if it supports basic Windows 7 gestures and has behavior and UI changes that make it more touch friendly. Upon request, Windows 7 will interpret touches as one of the basic gestures listed in table 4.2 and send the gesture events to the application.

Table 4.2 Windows 7 supports these basic gestures for "Better" applications.

Gesture Event	Description
Zoom	Indicates the start, progress, or end of a zoom gesture
Pan	Indicates the start, progress, or end of a pan gesture
Rotate	Indicates the start, progress, or end of a rotate gesture
Two finger tap	Indicates two fingers have tapped simultaneously
Press and tap	Indicates one finger is holding which a second finger tapped

These events are available to Win32 applications, and through some interop code, .NET applications as well. In the "Better" category, applications should also have interface elements sized appropriately for fingers, and all parts of the interface should behave properly with touch, including horizontal scroll areas.

These events allow for a basic touch experience, but it still is not the best. You cannot combine these basic Windows 7 gesture events, so panning and zooming at the same time is impossible, and you cannot pan two different things at the same time.

Gesture or Manipulation

The Windows 7 SDK describes these events as gestures. After reading chapter 2, you may be wondering if some of these are really manipulations. That is a good question. The gesture event only reports a single point that generally indicates the center of the gesture. Very carefully designed applications could use this information to create manipulations; however, that would require more work than creating a "Best" application.

These gesture events are designed so application developers can create a slightly better touch experience with minimal effort. The most common scenario for would be for developers to use the gesture point information for basic hit testing and use the gesture event to change a larger area of an application similar to how a scroll event affects the entire scrolling region.

There is a little grey area, since these events may have immediate feedback, but otherwise the gestures are not very direct. They're basically shortcuts for existing UI features, like a "Rotate 90 degrees" button or a "Zoom" button. In any case, this book is focused on "Best" applications, so debating the definitions of the basic gesture events is not necessary.

"Better" applications are typically GUI applications, but may have a modern design and may even be promoted as full touch applications. The experience is better than a "Good" application, but not nearly as good as a "Best" application.

BEST

Best applications make full use of the touch APIs, including raw touch events, manipulations, and inertia. These applications are designed for touch and have the best touch experiences. Natural user interfaces fit into the "Best" category and if you follow the guidance in this book, then you will be creating applications in the "Best" category.

NOTE

I would not disagree with you if you said that Windows 7 itself is still best used with a keyboard and mouse. It is still very much a graphical user interface and thus intended to be used with a mouse. The Windows 7 shell interface does rearrange itself subtly though when used on a touch-capable computer. Several aspects of the interface, such as the image viewer, use the native Windows 7 gestures. Windows 7 itself would fall into the "Better" category.

Now that it is clear that we want to create the best "Best" applications on Windows 7, you are probably wondering what specific APIs Windows 7 has for touch. There are actually several. We are only going to be focusing on WPF 4 for now, but let's make sure we're on the same page regarding the different APIs.

4.2.2 Windows 7 Touch APIs

For us, the most important feature of Windows 7 is the new touch APIs. There are several options that cover the entire range of Windows development, from native to managed to web. Let's quickly go over them.

WIN32 TOUCH API

Windows 7 comes with a native Win32 Touch API that includes support for raw touch events, several common gestures, as well as manipulations and inertia. The Win32 Touch API and native development is out of the scope of this book, but you can find more information on how to use the native Win32 Touch API on MSDN: http://msdn.microsoft.com/en-us/library/dd562197%28VS.85%29.aspx.

WPF 3.5 SP1 TOUCH INTEROP LIBRARY

WPF 3.5 does not includes any touch API, but if you need to you can develop multi-touch applications on Windows 7 using WPF 3.5 SP1 using the Windows 7 Multitouch .NET Interop Sample Library, located here: http://code.msdn.microsoft.com/WindowsTouch. This library is a bridge to the native Win32 touch API. I will not be talking about using this library in this book because WPF 4 includes a proper managed touch API.

WPF 4 TOUCH API

WPF 4 provides a built-in managed touch API, which is what we are going to focus on for now. This API was based upon the Microsoft Surface SDK, and is integrated with core WPF classes such as UIElement. All of the WPF 4 controls are touchable through touch promotion, but only the ScrollViewer include specific enhancements for touch. For fully touch-enhanced controls, you must turn to the Microsoft Surface SDK.

MICROSOFT SURFACE SDK

Microsoft Surface SDK is an additional Microsoft software release that contains touch-enhanced controls, templates, and samples. The difference between the controls built in to WPF 4 and the touch-enhanced controls in Surface SDK is that the SDK controls are built specifically for touch as the primary interaction mode. There are also specialized controls designed for certain multi-touch activities. I will introduce Surface SDK more in Chapter 5.

At this point, you know everything you need to know about the hardware and the Windows 7 multi-touch platform. It's time to make sure your development environment is set up and ready to go.

4.2.3 Setting up your development environment

We are going to be using WPF 4 for our multi-touch development. Getting set up for multi-touch development with WPF 4 is pretty easy. You will need to have the following software installed:

- Windows 7, any edition
- Visual Studio 2010, any edition
- Surface SDK 2.0

If you need a detailed walk-through on installing this software, take a look appendix A. Once it is installed, you should be good-to-go on the software development side of things. Since we are doing multi-touch development, though, you are going to need a multi-touch display or tablet that supports Windows 7 touch.

NOTE

Windows 7 depends upon the touch hardware drivers to generate proper touch messages. These messages are exposed as touch events in the Win32 API and WPF 4 API. There are some non-display multi-touch graphics tablets and laptops have multi-touch trackpads that interpret multi-touch through a custom API but don't send touch messages to Windows 7. For the purpose of this book, these devices are not useful. When evaluating multi-touch hardware, make sure to verify that the hardware works with native Windows 7 touch. The easiest way to test is to open Microsoft Paint and attempt to draw two or more lines at once.

If you do not have any multi-touch hardware available, then you can also emulate multi-touch input with special software.

4.2.4 Emulating Multi-Touch

If you do not have a multi-touch screen, then you will need to set up emulate touch input. You can do this with third-party software, such as Multi-Touch Vista. I will discuss another option, creating custom touch input devices, in Chapter 11.

Multi-Touch Vista is a free, open source (GPL) project that acts as a Windows 7 touch driver and provides input from a variety of devices, including multiple mice and input providers that support the Tangible User Interface Object (TUIO) protocol. Even though it has Vista in the name, it is designed for Windows 7 as well. To get started, go to http://multitouchvista.codeplex.com and download the latest release from the Downloads tab, shown in figure 4.7. Installation instructions and tutorial videos are available on the main page, and you can also read how to install this software in appendix A.

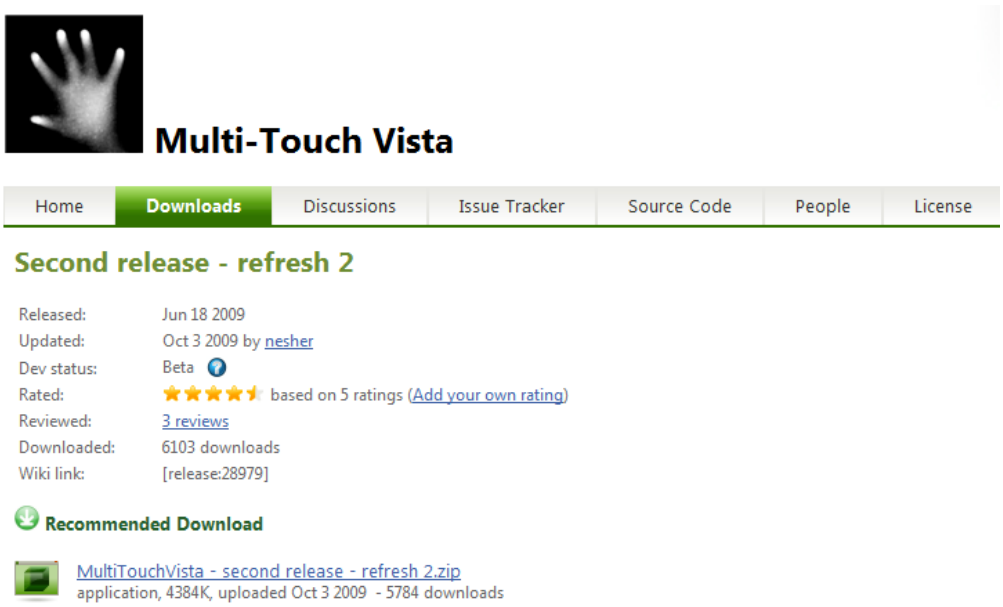


Figure 4.7 Download Multi-Touch Vista from http://multitouchvista.codeplex.com if you need to emulate multi-touch input with multiple mice.

Once Multi-Touch Vista is installed and started, you should be able to plug in multiple mice to your computer and use them to send touch events to Windows 7 as if you had real multi-touch hardware. You are now ready to start developing your first multi-touch application!

4.3 Hello, Multi-Touch

Now that you have everything installed, let's create your first program. There are only a few API features you'll need initially, so I will introduce them along the way. I will review these API features in detail in chapter 7.

ACTIVITY

This program will be a quick and dirty introduction to multi-touch manipulations. You will create a simple ScatterView-style program, which allows you to pan, scale, and rotate a rectangle.

4.3.1 Start a new project

On your multi-touch development machine, open Visual Studio 2010. Once Visual Studio has loaded, start a new project from either File | New | Project… or by clicking on New project on the Start Page, shown in figure 4.8.

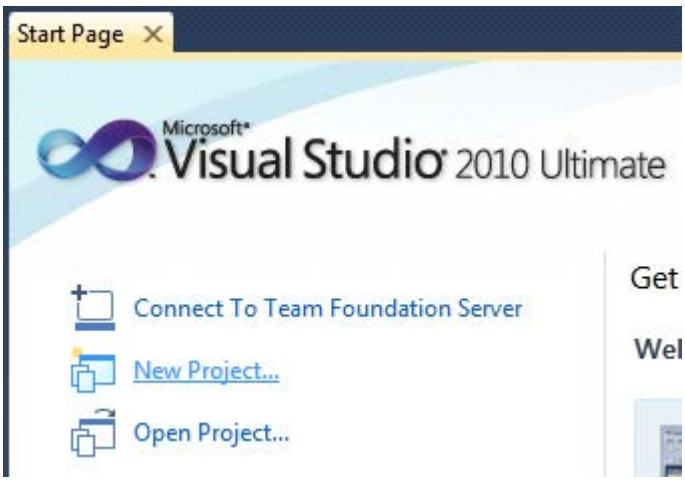


Figure 4.8 You can't start developing your first multi-touch application until you click New Project… on the Visual Studio 2010 Start Page.

In the New Project dialog, select Visual C#, then Windows. On the right, select WPF Application, type "HelloMT" for the name, then press OK. Double-check figure 4.9 if you are lost at this point.

NOTE

Everything in this book can be done with both C# and Visual Basic. For brevity, I will only show C# examples in the book. The downloadable code samples contain both C# and Visual Basic version of each code sample.

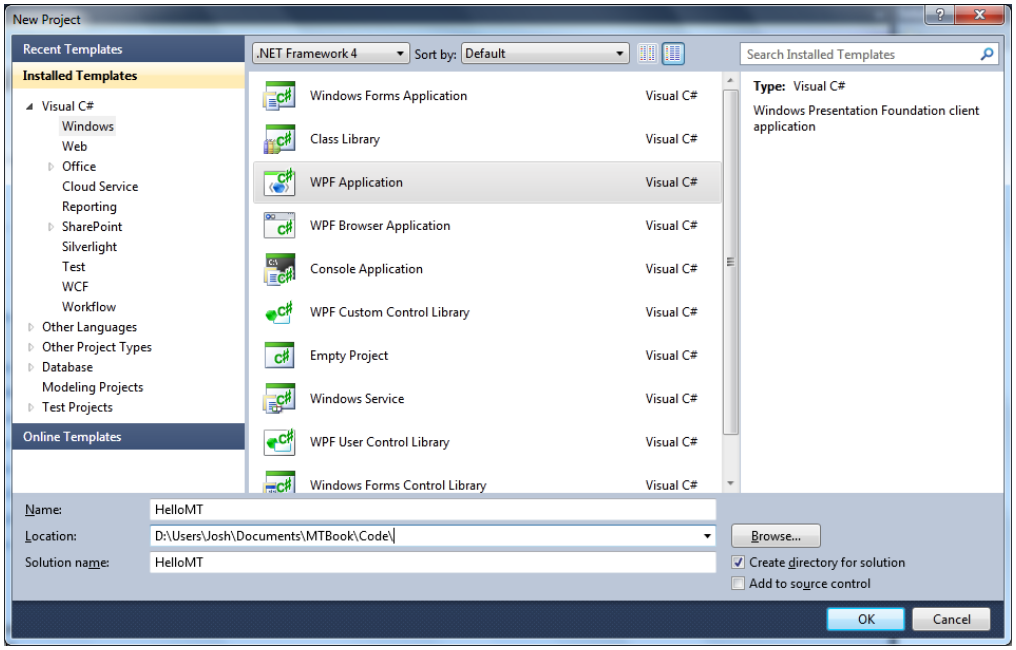


Figure 4.9 Veteran software developers know the New Project dialog well. For your first multi-touch application, select the humble WPF Application template, and name it HelloMT.

Now you have a basic WPF project created, ready to be turned into a great natural user interface, or at least a Hello, Multi-touch sample. Let's start adding code.

4.3.2 Basic interface visual

In the Solution Explorer, double click on MainWindow.xaml to open the XAML file. You'll see the following default code:

```
<Window x:Class="HelloMT.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

First, change the `Width` and the `Height` properties of the `Window` to 800 and 600, respectively. I also like to give my test applications a fixed `Left` and `Top`, so set the `Left` and `Top` each to 10. Finally, change the `Title` to "HelloMT".

```
<Window x:Class="HelloMT.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="HelloMT"
        Width="800" Height="600"
        Left="10" Top="10">
```

Then replace the `<Grid> </Grid>` with the following:

```
    <Canvas>
        <Border Width="300"
                Height="300">
            <Border.Background>
                <LinearGradientBrush>
                    <GradientStop Color="Red"
                                  Offset="0" />
                    <GradientStop Color="Blue"
                                  Offset="1" />
                </LinearGradientBrush>
            </Border.Background>
        </Border>
    </Canvas>
```

This just creates a `Border` with a gradient background. If you run the program, all you will see is a colored rectangle in the corner of the window.

Boring! Let's add some interaction!

4.3.3 Enable manipulations and add panning

Here comes the *WOW* moment. All you need to do to receive touch events is set a property and add an event handler. On the `Border` element, add this:

```
        <Border Width="300"
                Height="300"
                IsManipulationEnabled="True"
                ManipulationDelta="Border_ManipulationDelta">
```

The `IsManipulationEnabled` part is a new property on `UIElement`. When set to true, that `UIElement` will capture touch contacts and track manipulations, calling the appropriate Manipulation event handlers. One of those is `ManipulationDelta`, which is called every time a tracked contact changes.

Open MainWindow.xaml.cs from Solution Explorer. Add the code in listing 4.1 to set up the `ManipulationDelta` event handler. This code gets a reference to the element that triggered the event and then creates and updates a `RenderTransform` based upon how the manipulation has translated. Warning: this code will purposely behave badly, and we'll fix it as soon as we see the behavior and discuss the bug.

Listing 4.1 Translate an UIElement using a manipulation

```
private void Border_ManipulationDelta(object sender,
                                      ManipulationDeltaEventArgs e)
{
    FrameworkElement element =                                          #A
                     sender as FrameworkElement;
```

```
        if (element != null)                                                #B
        {
            Matrix matrix = new Matrix();                                   #C

            MatrixTransform transformMatrix =                               #D
                            element.RenderTransform as MatrixTransform;

            if (transformMatrix != null)                                    #1
            {
                matrix = transformMatrix.Matrix;                            #E
            }

            matrix.Translate(                                           #F
                            e.DeltaManipulation.Translation.X,
                            e.DeltaManipulation.Translation.Y);

            element.RenderTransform =                                   #G
                                new MatrixTransform(matrix);

            e.Handled = true;                                           #H
        }
    }
```

#A Get who triggered the event
#B Make sure this is a FrameworkElement
#C Initialize a blank Matrix
#D Try to get the existing MatrixTransform
#1 If it exists,
#E Then retrieve the matrix from it
#F Apply the delta translation
#G Update with the transformed matrix
#H We took care of the event

This code assumes that the first time it is called it must create a transformation matrix. This transformation matrix is stored in the element's `RenderTransform` property. During subsequent calls, it will attempt to retrieve the transformation matrix #1 and then update it.

Go ahead and run the program now. Touch the rectangle and drag it around. How does it feel? It's really jumpy, isn't it? Not smooth at all. There's a good reason for that, and an easy fix!

Here's the problem: When you move your finger, it triggers the `ManipulationDelta` event, which translates the `Border` by the `DeltaManipulation`. The `DeltaManipulation` contains the relative movement of the finger to the `Border` since the last time the `ManipulationDelta` event handler was called. The next finger movement, though, takes into account both the new finger movement and the translation of the `Border`. This adds up, then the `Border` starts flickering back and forth, and then events are even triggered when the finger is holding still due to the `Border` translation. This all causes the flicker. Figure 4.10 illustrates the situation leading to manipulation flickering.

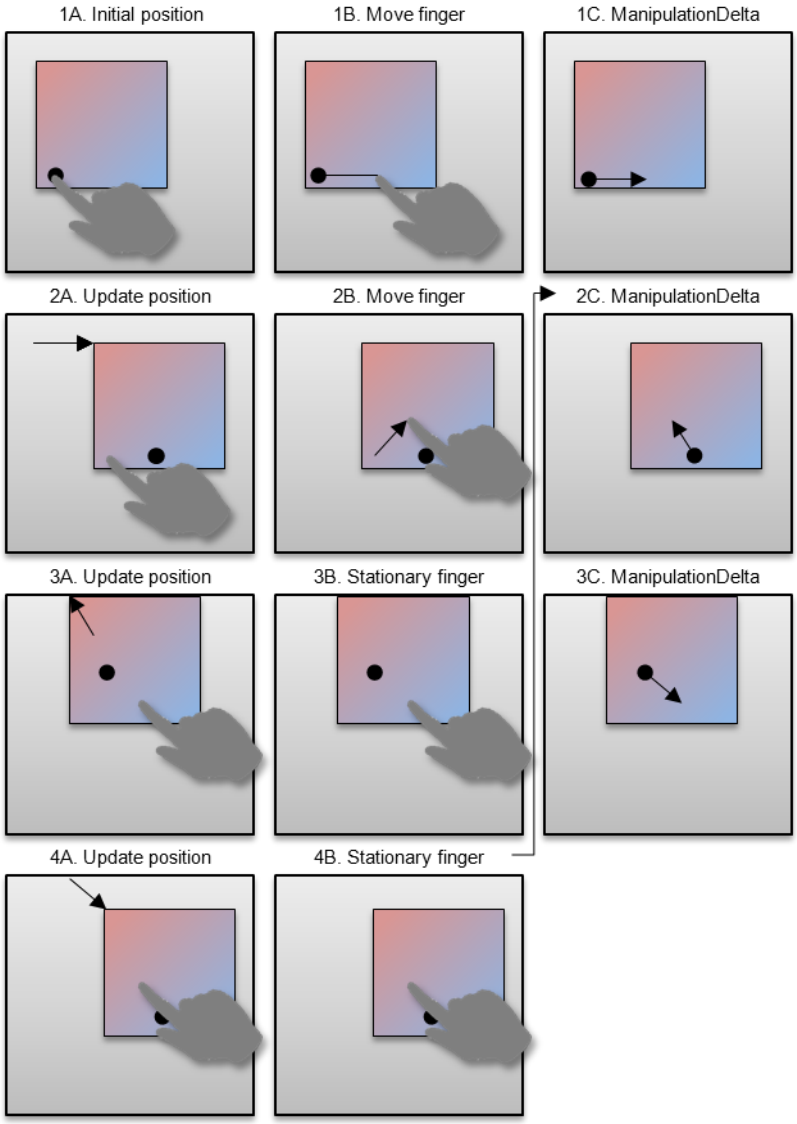


Figure 4.10 When a pan manipulation is relative to the object being panned, flickering will occur. The new object position is calculated using the position of the finger at the last `ManipulationDelta`, shown by the dot. After 4B, the sequence will repeat to 2C until the finger is lifted.

The core problem is the fact that the manipulation is calculated relative to the `Border`, which is a moving target. The solution is to change what the manipulation is relative to. We can do that in the `ManipulationStarting` event handler.

In MainWindow.xaml, set the `ManipulationStarting` event and add a name to the main `Canvas` element:

```
<Canvas Name="LayoutRoot">
    <Border Width="300"
            Height="300"
            IsManipulationEnabled="True"
            ManipulationDelta="Border_ManipulationDelta"
            ManipulationStarting="Border_ManipulationStarting">
```

Then in MainWindow.xaml.cs, add the following event handler for `ManipulationStarting`. This line simply assures that the manipulation is calculated from the parent `Canvas`.

```
private void Border_ManipulationStarting(object sender,
                                         ManipulationStartingEventArgs e)
```

```
{
    e.ManipulationContainer = LayoutRoot;
}
```

Now run the program again. What do you think now? Much better! The rectangle, shown in figure 4.11, pans smoothly and has no flicker.

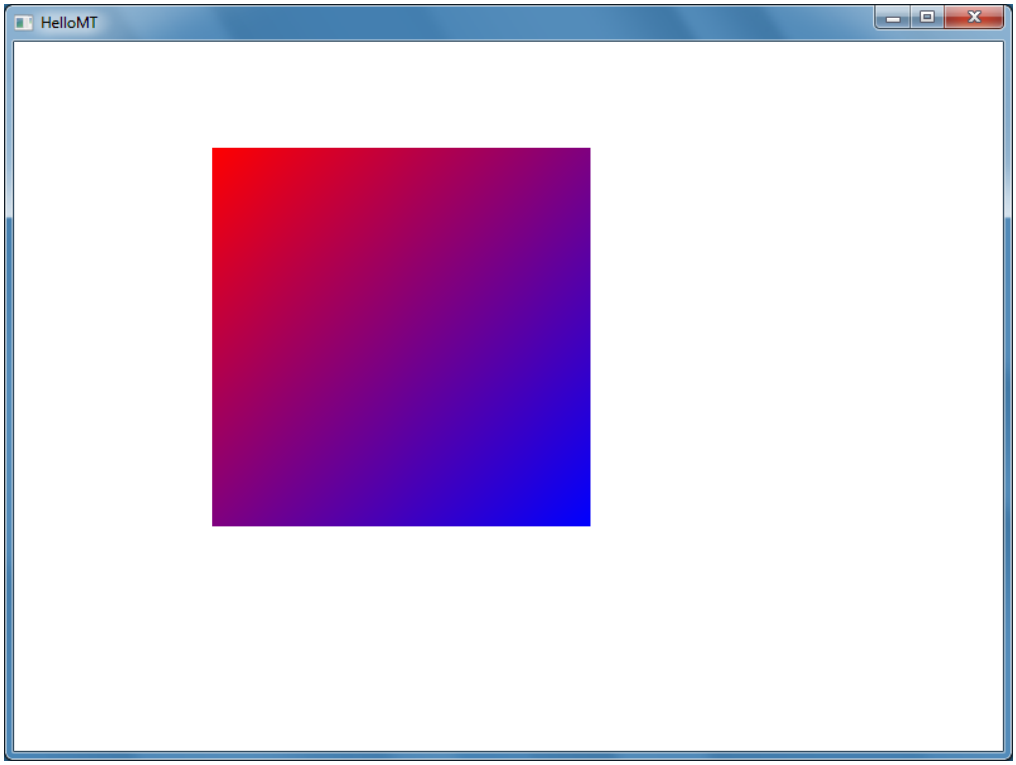


Figure 4.11 HelloMT now has smooth panning manipulations

Panning alone was pretty easy, but with the framework we have already set up, we can add a few more lines of code to enable more advanced manipulations such as scaling and rotation.

4.3.4 Add scaling and rotation

You have the basic program, and with a few more lines we have add both scaling and rotation. In MainWindow.xaml.cs, add the bolded code in listing 4.2 to the `Border_ManipulationDelta` method. I provided the previous and next lines so you will know where to insert the code.

Listing 4.2 Enable scaling and rotation

```
matrix.Translate(e.DeltaManipulation.Translation.X,
                 e.DeltaManipulation.Translation.Y);

Point transformCenter =                                          #A
                      new Point(element.ActualWidth / 2,
                                element.ActualHeight / 2);

matrix.RotateAt(e.DeltaManipulation.Rotation,                    #B
                transformCenter.X,
                transformCenter.Y);

matrix.ScaleAt(e.DeltaManipulation.Scale.X,                      #C
               e.DeltaManipulation.Scale.Y,
               transformCenter.X,
               transformCenter.Y);
```

```
element.RenderTransform = new MatrixTransform(matrix);
```
#A Find the center of the element
#B Apply the delta rotation at the center
#C Apply the delta scale at the center

This adds basic scaling and rotation and it will work if you run it, but there's something not quite right. Go ahead and play with it for a few minutes before continuing reading.

Here's a hint: expand the rectangle to the full size of the window, then put a finger on the top right and bottom right corners, then rotate your fingers. Try the same thing again and pinch your fingers together. Figure 4.12 illustrates the problem.

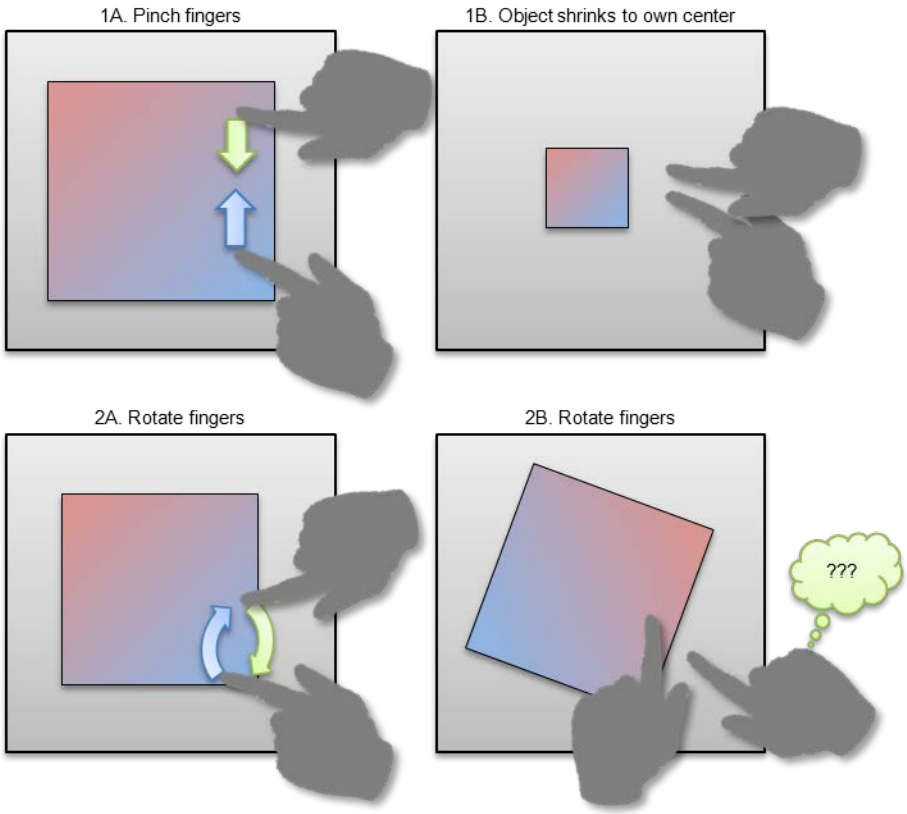


Figure 4.12 Scaling and rotating around the object center instead of the manipulation origin can cause unnatural behavior. In this case, if the manipulation is not centered on the object, then the object will not track with the fingers.

The problem is that the rectangle does not stay under your fingers as you manipulate it. It rotates and scales around its own center, which is fine when your fingers are centered on the rectangle, but that often is not the case.

Manipulations and ManipulationOrigin

Manipulations track all of the fingers that touched a user interface element until those fingers are released. One property that is a member of `ManipulationDeltaEventArgs` is `ManipulationOrigin`. This is calculated as the average position of all tracked fingers that are still down.

In the ideal interaction, when your fingers touch various points on this rectangle, those points will stay under your fingers at all times. We can do this by changing the `transformCenter`. In MainWindow.xaml.cs, replace this line:

```
Point transformCenter = new Point(element.ActualWidth / 2,
                                  element.ActualHeight / 2);
```

with these two:

```
Point transformCenter = LayoutRoot.TranslatePoint(                #A
                            e.ManipulationOrigin,
                            element);

transformCenter = matrix.Transform(transformCenter);              #B
```

#A Get ManipulationOrigin relative to element
#B Transform it by the current matrix

Those lines use the `ManipulationOrigin` as the transformation center point for scaling and rotation. Since we previously specified that the manipulation is relative to `LayoutRoot`, we must first get the `ManipulationOrigin` point relative to `element`. Then we transform the point with the matrix to account for the existing rotation and translation from previous `ManipulationDelta` events.

Now try the same tests from before and see how the behavior is different. It should feel much more natural. You should see a behavior similar to figure 4.13.

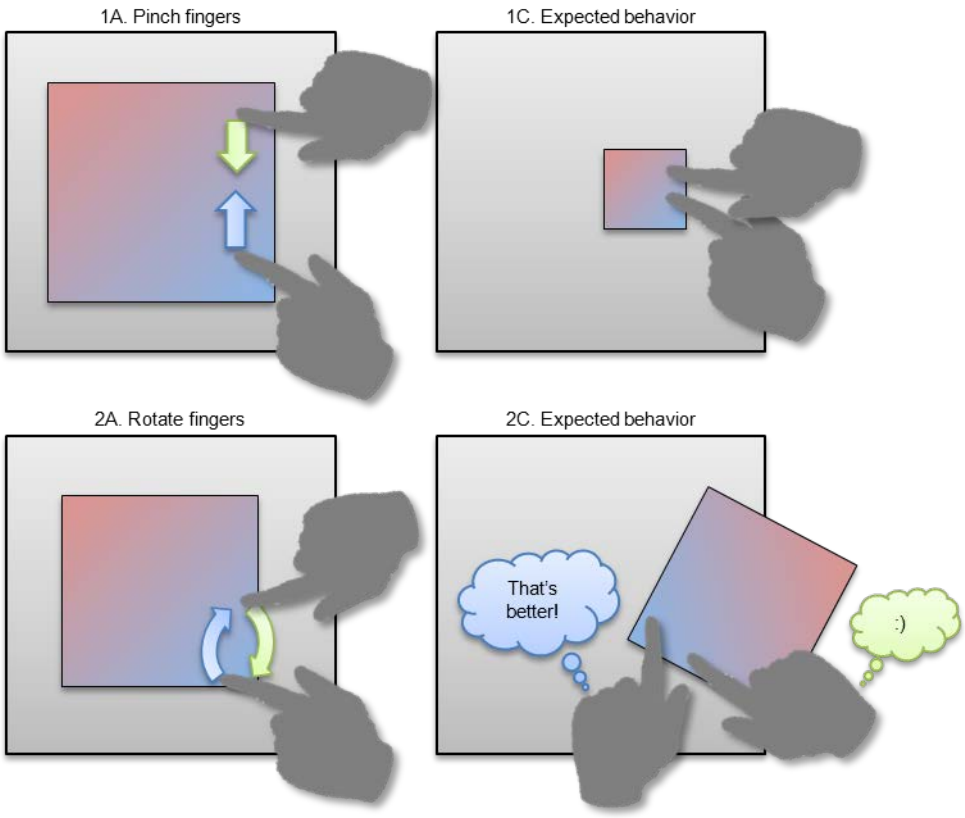


Figure 4.13 Now that the object transformation follows the manipulation origin, the object tracks the fingers while panning, rotating, and scaling. This behavior is much more natural.

There is one more thing that I'd like to add to HelloMT to make it even more natural feeling: inertia.

4.3.5 Add inertia

Inertia will let us "throw" objects. If we are panning an object and lift our finger before stopping, the object will continue to move for a short period of time. This is similar to the real world and is one more thing that can make objects seem more real.

We can add inertia with one more event handler – `ManipulationInertiaStarting`. This event is called whenever all of the fingers being tracked have lifted. It is used to set up the desired inertia parameters, if any. Afterwards, `ManipulationDelta` events continue to be called even though all the fingers are lifted, until the inertia runs out. We will see in later chapters how the desired inertia parameters may change with every manipulation, depending upon the final state. For now, we will set a simple deceleration.

By default, inertia is off, so as soon as you release your fingers the rectangle stops. Let's change that behavior by adding the `ManipulationInertiaStarting` event to the `Border` element in MainWindow.xaml:

```
<Border Width="300"
        Height="300"
        IsManipulationEnabled="True"
        ManipulationDelta="Border_ManipulationDelta"
        ManipulationStarting="Border_ManipulationStarting"
        ManipulationInertiaStarting="Border_ManipulationInertiaStarting">
```

Then add following code to the event handler in MainWindow.xaml.cs. This will tell WPF how quickly we want to decelerate the translation and rotation. Decelerations are specified in Device Independent Pixels (DIP) per millisecond and degrees per millisecond.

```
private void Border_ManipulationInertiaStarting(object sender,
                          ManipulationInertiaStartingEventArgs e)
{
    e.TranslationBehavior.DesiredDeceleration = .001;            #A
    e.RotationBehavior.DesiredDeceleration = .001;               #B
}
#A DIP/millisecond
#B degrees/millisecond
```

Now try it out! Lift your finger while moving the rectangle and it will keep going for a while. Try lifting your fingers while you are rotating. You may have to practice this flicking type motion because the natural tendency is to stop our fingers before lifting.

Be careful though – you will probably accidentally fling the rectangle outside the window, never to return. If that happens, you'll have to close the program and run it again.

Before we enabled inertia, when you lifted your fingers the rectangle would stop immediately and the worst you could do is push it close to the edge of the window. With inertia, we have to be aware of the borders and the possibility of the user losing the objects. Let's add a simple check to stop the inertia if the rectangle hits a border. In MainWindow.xaml.cs, add the bolded code in the snippet below near the bottom of `Border_ManipulationDelta`.

```
element.RenderTransform = new MatrixTransform(matrix);

Rect containerBorder = new Rect(LayoutRoot.RenderSize);              #A

Rect elementBorder =                                                 #B
                    element.RenderTransform.TransformBounds(
                        new Rect(element.RenderSize));

if (e.IsInertial &&                                                  #C
    !containerBorder.Contains(elementBorder))                        #C
{
    e.Complete();                                                    #D
}

e.Handled = true;
```

#A Rect of the container
#B Transformed rect of the element
#C If not fully in container
#D Complete the inertia

This does a basic boundary check to make sure that the element is fully within the container, in this case `LayoutRoot`. The property IsInertial is true only if the manipulation has entered the inertia phase. If the manipulation is currently in inertia mode and the element is not contained, then `e.Complete()` will end the manipulation, including inertia, and stop the rectangle.

Test the program again, and you will see that the rectangle does stay within the borders if you try to throw it out. You can still drag it close to the border of the window though.

NOTE

All sample code is downloadable from this book's website: www.HandsOnNUI.com. The complete HelloMT program as of this point is available in \Chapter 1\HelloMT\.

Congratulations! You've created your first multi-touch application. It is quite simple, but in the process you've learned a lot of the basics about manipulations and some ways to make the manipulation work naturally. This is just the first step, though. In the next few chapters, we are going to build upon this knowledge and learn more how to use the WPF 4 Touch API to create more functional controls and interfaces with very natural behaviors.

4.4 Summary

This chapter focused on getting you started with multi-touch development. We learned about the different layers of abstraction that lead from the detection of touch points to your application.

We discussed types of multi-touch hardware and how the behavior and limitations of the hardware can affect your application design. The way you design interactions can be limited or enhanced by the number of touches that the hardware supports. The hardware reports touches to the Windows 7 multi-touch platform, which has several different multi-touch API options. In this part, we are focusing on WPF 4.

To get you started quickly with the WPF 4 Touch API, we walked through a very simple application that allows you to pan, scale, and rotate a rectangle using multi-touch manipulations. This program recreates the behavior of the ScatterView control and gave you a glimpse behind the scenes of how ScatterView works. In most cases, you won't need to recreate these common behaviors because the Surface SDK provides them for you.

In chapter 5, we will update the HelloMT sample to use a proper ScatterView, and then talk about the Surface SDK, what it provides, and how you can use it to develop really advanced natural user interfaces very quickly.

5

Using traditional Surface SDK controls

In chapter 4, we spent a bit of time creating the HelloMT application. While it was an enlightening exercise, creating and testing the logic and math behind the behavior was relatively tedious. Worse, we did not even create something that is easily reusable. Fortunately, aside from the learning exercise or very advanced custom layout scenarios, we do not need to deal with these issues because for the most part, they have been solved for us and packaged into an easy to use form as the Microsoft Surface SDK.

Surface SDK is a collection of WPF controls and infrastructure provided by the Microsoft Surface team. This SDK enables you to easily build great multi-touch applications for Microsoft Surface hardware as well as standard touchscreens on desktops, tablets, and slates running Windows 7.

You will want to make full use of Surface SDK in your multi-touch WPF applications. It saves a significant amount of time that you would otherwise spend recreating common multi-touch controls and re-solving basic multi-touch challenges.

NOTE

This chapter is written against a pre-release version of Surface SDK 2.0. The final version may include changes and this chapter will be updated for the final release.

The controls in Surface SDK fall into two categories: existing controls updated for touch, and new controls designed specifically for touch. This chapter will cover the existing, or traditional, controls that are included in Surface SDK. The new controls will be covered in chapters 6 and 7.

Surface SDK also includes touch visualizations and a very important drag-and-drop framework, which I will describe only at a high level in this chapter. These features deserve their own chapter so it will be covered in detail in chapter 10 after we learn about WPF's raw touch API and manipulations API.

For now, let's introduce the full Surface SDK and get to know the features, benefits, and how to get started.

5.1 Surface SDK overview

The Surface SDK has many different components but they can be split into four main categories, shown in Table 5.1.

Table 5.1 Surface SDK consists of several different components that make it easier to develop high-end multi-touch experiences

Category	Description
Traditional controls	Standard WPF controls updated for proper touch behavior
Multi-touch controls	Entirely new controls useful in a multi-touch environment
Touch visualizations	Auras, trails, and tethers to improve the touch experience
Drag-and-drop framework	Rich asynchronous drag-and-drop capabilities

Although many natural user interface designs will minimize the use of traditional GUI controls, such as buttons, checkboxes, and radio buttons, they are sometimes still needed and are the best solution for some tasks. If you are going to use any traditional controls in your multi-touch application, then you're going to want to use the Surface SDK versions. The SDK provides touch optimized versions of traditional controls plus several multi-touch only controls.

NOTE

To clarify, I will refer to controls that come with WPF and the .NET Framework, such as Button, Checkbox, and ListBox, as native controls or native WPF controls. These are in contrast to the traditional Surface SDK controls, such as SurfaceButton, SurfaceCheckbox, and SurfaceListBox, which are touch-enhanced versions of the native controls. Multi-touch controls will always refer to the new Surface SDK controls designed specifically for touch.

In this section I will discuss some of the reasons why Surface SDK exists and the benefits of using Surface SDK.

MULTI-TOUCH CONTROLS FOR MULTI-TOUCH EXPERIENCES

The traditional and multi-touch Surface controls enable two different types of interactions that are not possible with the mouse-only native WPF controls. First, each control is multi-touch aware and behaves properly when you interact with the control using more than one finger. Second, you can interact with multiple Surface controls at the same time. The native WPF controls and Windows in general only allows interaction with one control at a time. As a side-effect some concepts, like the idea of control focus or tabbing to change the focus, do not make sense in a multi-touch application.

Interacting with multiple controls simultaneously is important for single users but especially for multiple users. I often let my young daughters "test" my multi-touch applications. As I watch them play with the applications, I've noticed that they do not have any concept of focus or modality. They see something and touch it and expect it to respond, regardless of the fact that someone else is interacting with an object somewhere else on the screen. This matches their expectations learned from interacting with real-world objects. Could you imagine the fights that would occur if their playroom only allowed one child to interact with one toy at a time with a single finger? The Surface controls are critical to making your applications more natural by enabling both multi-touch and multi-user scenario.

SURFACE SDK EXTENSIONS TO WPF

In addition to the controls, Surface SDK provides some extensions to the WPF infrastructure including touch visualizations and a drag-and-drop framework.

Touch visualizations help users see that the screen is recognizing and accepting touches. The Microsoft Surface team has done extensive user research in this area and found that touch visualizations improve the user accuracy when touching small visual targets. This improves the overall experience because it provides immediate feedback to the user to help them understand the current status of the interface.

One way touch visualizations do this is by changing the visualization to make visible hidden information about the interaction. For example, when a touch is captured to a control but the touch moves outside the control, a tether will be drawn between the touch and the control so the user understands there is a relationship between the two.

Surface SDK also provides a drag-and-drop framework that allows rich drag-and-drop operations between multi-touch controls. Although WPF and Windows already have drag-and-drop features, they are icon-centric and mouse-based and do not work well for the needs of a multi-touch natural user interface.

FASTER AND EASIER DEVELOPMENT

Multi-touch and NUI development can be challenging, sometimes because the entire interaction paradigm is new, and sometimes because we don't have a huge history of components and solutions yet that we have for GUI. Surface SDK addresses this problem.

> **NOTE**
>
> Surface SDK makes multi-touch development with WPF much easier. Unfortunately, it is not available for Silverlight.

The whole point of the Surface SDK is to help you create your multi-touch applications quicker and easier than without it. The SDK does not provide everything you could possible want, but it makes a lot of very common scenarios much easier.

WHAT IS NEW IN 2.0

This book targets Surface SDK 2.0, which was released along with the new Microsoft Surface 2.0 platform and hardware. Some of you may have heard of Surface SDK 1.0 or used it for developing for Microsoft Surface 1.0.

While Surface SDK 1.0 was only usable for the Microsoft Surface hardware running Windows Vista, Surface SDK 2.0 can be used to develop multi-touch NUIs across any Windows 7 computer with a touchscreen. This includes the Surface 2.0 hardware as well as tablets and slates running Windows 7 and Windows 7 desktops that are connected to multi-touch displays.

> **NOTE**
>
> From a technical point of view, applications written with Surface SDK 2.0 can run on computers without multi-touch screens. If you are targeting both multi-touch and mouse-only computers, you'll need to be careful about making sure the computers without multi-touch can still access the necessary features of your application. I don't recommend targeting computers without multi-touch screens though. That would force you into compromises where the interface will be worse for everyone compared to designing a multi-touch interface and a separate mouse interface.

Besides running on any Windows 7 machine, Surface SDK 2.0 also features an updated look-and-feel for the controls and takes advantage of the WPF 4 touch APIs. The WPF 4 touch APIs were based upon Surface SDK 1.0 features but are now integrated into the WPF infrastructure, providing a more consistent and easier to use development experience.

If you haven't already, you'll need to install Surface SDK on your development computer. See Appendix A for installation instructions.

I mentioned that one of the main advantages of using Surface SDK is that it makes development easier and quicker. To prove the point, let's revisit the HelloMT sample from chapter 4 and see how the ScatterView control makes everything better. Once we see that, we'll jump into the meat of this chapter and learn about the traditional Surface SDK controls that were updated for touch.

5.1.1 Hello, Surface SDK

Surface SDK can save you a lot of effort. In chapter 4 we spent a bit of time creating the relatively common multi-touch behavior of the ScatterView. We can rewrite the entire HelloMT sample using the ScatterView control with no code behind and at the same time get some extra features that come with the ScatterView control.

The ScatterViews automatically provide visual cues such as shadows and animation to help the user understand when they have touched and activated a ScatterViewItem, which is the container for content in a ScatterView. In addition, we automatically get Surface-style touch visualizations because we used

SurfaceWindow instead of the regular WPF Window. Figure 5.1 shows the HelloSurfaceSDK program with a ScatterViewItem shadow and two touch visualizations.

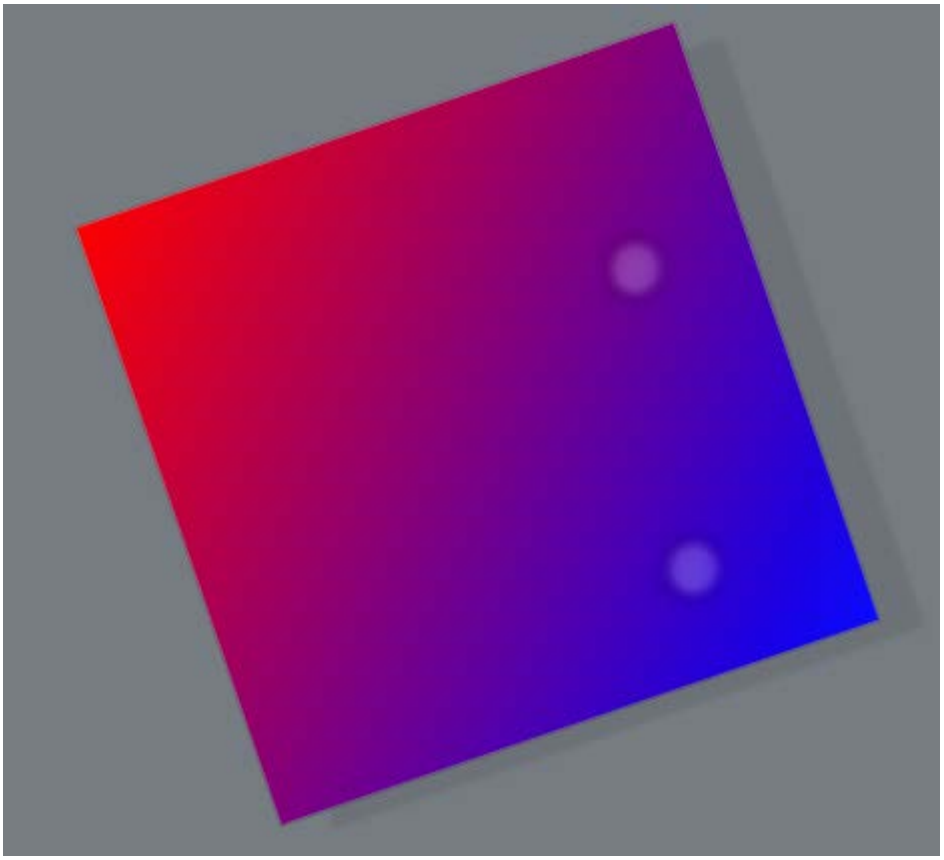


Figure 5.1 The HelloSurfaceToolkit sample shows how Surface SDK's ScatterViewItems provide shadows and the SurfaceWindow provides touch visualizations. These are both subtle but important for helping the user understand what is going on with the interaction.

The HelloSurfaceSDK sample is just a panning, zooming, rotating rectangle just like the HelloMT sample from chapter 4. The main difference is the use of ScatterView and no code necessary. The XAML for HelloSurfaceSDK is shown in listing 5.1.

Listing 5.1 HelloSurfaceSDK uses ScatterView in XAML with no code

```
<s:SurfaceWindow x:Class="HelloSurfaceSDK.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:s="http://schemas.microsoft.com/surface/2008"
        Title="HelloSurfaceSDK">
    <Grid>
        <s:ScatterView>                                          #1
            <s:ScatterViewItem Width="300"                       #1
                               Height="300"
                               Center="150, 150"
                               Orientation="0">
                <Viewbox>
                    <Rectangle Name="rect"
                       Width="300"
                       Height="300"
                       >
                        <Rectangle.Fill>
```

```
                        <LinearGradientBrush>
                            <GradientStop Color="Red" Offset="0" />
                            <GradientStop Color="Blue" Offset="1" />
                        </LinearGradientBrush>
                    </Rectangle.Fill>
                </Rectangle>
            </Viewbox>
        </s:ScatterViewItem>
    </s:ScatterView>
  </Grid>
</s:SurfaceWindow>
```

#1 New controls from Surface SDK

We achieved the same result as HelloMT by putting the same rectangle inside of the ScatterViewItem, which is inside of a ScatterView. Using these controls saves several hundred lines of code and significant development and testing time, once you factor in the extra features.

The ScatterView control falls into the "multi-touch controls" category and we will be covering it in detail in chapter 6. There are also several other controls designed specifically for touch that we will discuss in chapter 7.

While it makes sense for Surface SDK to include entirely new multi-touch controls like ScatterView, you might wonder why it includes equivalents of the native WPF controls. WPF provides a whole library of controls and they respond to touch, right?

Sort of. To get the full story, let's spend a little time discussing the native WPF controls. Later in this chapter, we will explore the updated controls that WPF provides.

5.2 Native WPF controls

WPF provides many common controls and they will respond to touch, but not always in the way you expect. The native WPF controls were designed for a mouse-based interface. This means the control logic is configured for a single input point and they have visual components that are too small to reliably hit with a finger on a touchscreen.

Touching native WPF controls

With one exception, native WPF controls only respond to the first touch in the window. This is because the first touch, unless handled by the touch API, will be promoted to a mouse event. The first touch is also referred to as the primary touch point. (We will discuss mouse promotion more in section 8.6.) Since the mouse system can only handle a single point, it will only promote the primary touch point and subsequent touches are ignored until all touches are raised and a new primary touch point is created.

This setup has advantages in some cases. Applications in the Windows 7 "Good" touch category, discussed in section 4.2.1, make use of this automatic mouse promotion. These GUIs that are unaware of touch will still operate on a touch screen, although only as a single touch. The touch experience will not be very good, though.

To provide one example, consider the native WPF button. If you touch it with two fingers, then let go with one finger, the Click event is fired even though the second finger is still touching. This violates most users' expectations based upon how real life buttons work. In addition, you cannot interact with multiple native WPF controls at the same time.

There is only one native WPF control that is aware of touch at all: the ScrollViewer. The ScrollViewer control has been enhanced to enable smooth touch scrolling and it behaves well with multiple touches.

There is also a SurfaceScrollViewer found in Surface SDK, which is based upon the native ScrollViewer. Before we get to the Surface SDK controls, I think it will be helpful to learn about the native ScrollViewer. In the rest of this section I'll discuss the touch enhancements to the ScrollViewer and give an example program of how to use these new features.

5.2.1 ScrollViewer enhancements

In WPF 4, if you use a regular ScrollViewer, or use a control like ListBox or TextBox which are based upon ScrollViewer, then you will just get a regular ScrollViewer and nothing special regarding touch. If you set the PanningMode property of the ScrollViewer, though, you will be able to touch and pan the contents of the ScrollViewer.

> **NOTE**
>
> The scrollbar, scroll thumb, and arrows are not included in the touch enhancements. Those elements are like the rest of the native controls and only respond to the first touch.

There are three new properties WPF 4 adds to ScrollViewer, including PanningMode, PanningDeceleration, and PanningRatio. These properties are also available on the SurfaceScrollViewer in the Surface SDK. Table 5.2 describes these properties.

Table 5.2 WPF 4 adds several properties to ScrollViewer to enhance the touch experience

Property	Default value	Description
PanningMode	None	How the ScrollViewer responds to horizontal and vertical panning
PanningDeceleration	0.001	Used in flicking inertia. Determines how quickly the scrolling slows after a flick. Units are device-independent units per squared millisecond and the value must be greater than or equal to zero.
PanningRatio	1.0	Ratio between finger movement and scrolling movement. Values greater than 1 make scrolling faster than fingers. Values between 0 and 1 make scrolling slower than fingers. Value must be greater than or equal to zero.

Since PanningMode defaults to None, there is no touch scrolling initially. The PanningMode determines whether or not horizontal or vertical panning is captured for touch scrolling or promoted to mouse events. If you want touch scrolling you will need to set it to an appropriate PanningMode value, enumerated in table 5.3.

Table 5.3 The ScrollViewer's has several possible PanningModes

PanningMode	Description
None	(Default) No touch scrolling, first touch promoted to a mouse event
HorizontalOnly	Horizontal panning causes scrolling, vertical panning is ignored
VerticalOnly	Vertical panning causes scrolling, horizontal scrolling is ignored
Both	Both horizontal and vertical panning is enabled
HorizontalFirst	If the first movement is horizontal, then scrolling in both directions is possible. Otherwise, the first touch is promoted to a mouse event
VerticalFirst	If the first movement is vertical, then scrolling in both directions is possible. Otherwise, the first touch is promoted to a mouse event

The PanningModes only determine how the touches are interpreted. You will still need to set the horizontal and vertical scroll bar modes, as appropriate. For example, if PanningMode is set to VerticalOnly and VerticalScrollBarVisibility is set to disabled, then you will have no touch scrolling. Internally, it is

tracking the vertical component of the manipulation but since the vertical scroll bar is disabled it does not allow scrolling to actually occur.

In the same way, if you want horizontal touch scrolling, you will need to enable the horizontal scrollbar. You can always have one or both scrollbars set to hidden, which will allow touch scrolling without the visible scrollbars.

The VerticalFirst mode is particularly important for situations where you will be scrolling through text but still want to be able to select or highlight text. This behavior is common in web browsers. If you move your finger vertically at first, then the page will scroll. If you move your finger horizontally first, then the touch point will be promoted to a mouse event. In this case, just like a regular mouse cursor you will start selecting text.

Let's take a look at a sample application that shows the different behaviors of the ScrollViewer. We will explore the touch scrolling settings and give you a chance to get to know how it behaves in different scenarios.

5.2.2 TouchScrolling sample application

This sample is called TouchScrolling and is available with the book sample code. Figure 5.2 shows what the TouchScrolling application looks like.

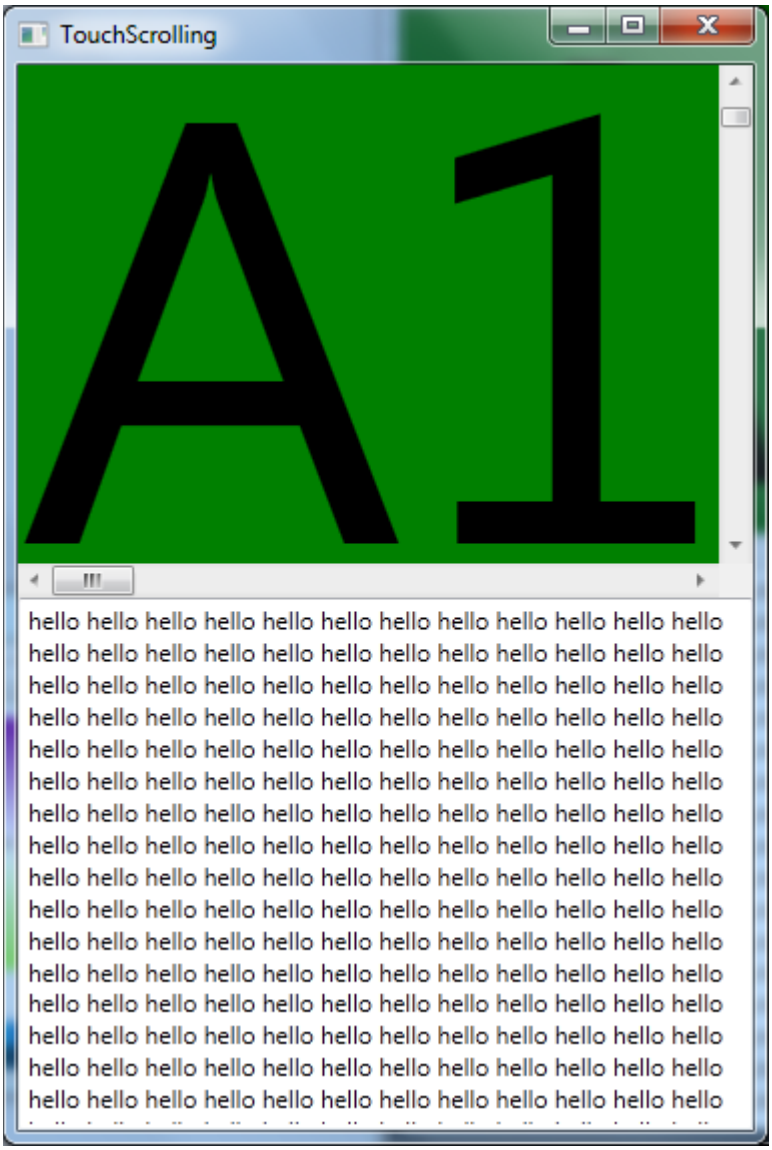


Figure 5.2 The TouchScrolling sample application lets you experiment with different panning options available with a ScrollViewer. The top half of the window is a simple ScrollViewer while the bottom half is a TextBox, which has a ScrollViewer in the control template. The different panning options let you control when and how the ScrollViewer responds to touch and when the touches are allows to promote to mouse events to allow, for example, text selection within the TextBox.

If you open the project, you will see a single window that includes the following XAML in listing 5.2.

Listing 5.2 The TouchScrolling sample includes the following XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <ScrollViewer Background="Green"
                  PanningMode="Both"                          #A
                  PanningDeceleration="0.001"                 #A
                  PanningRatio="1"                            #A
                  HorizontalScrollBarVisibility="Auto"
                  VerticalScrollBarVisibility="Auto"
```

```
                Grid.Row="0">
        […content…]
    </ScrollViewer>
    <TextBox ScrollViewer.PanningMode="VerticalFirst"              #1
             Grid.Row="1"
             TextWrapping="Wrap">
        […content…]
    </TextBox>
</Grid>
```

#A New touch properties
#1 Configuring TextBox's internal ScrollViewer

Notice that you can set the new panning properties directly on the ScrollViewer as well as on a TextBox, which has a ScrollViewer in its control template #1. Of course, since the PanningMode property doesn't belong to TextBox, you need to use the ScrollViewer.PanningMode syntax so it will work property as an attached property.

NOTE

The default PanningMode for TextBox is VerticalFirst so setting the property in this example #1 is redundant. It is included to illustrate the syntax necessary to customize this behavior. Some other native controls that include a ScrollViewer also set a default. For example, the default PanningMode for a ListBox is Both.

Go ahead and play around with the properties. Try changing the PanningMode to different values and testing what happens when you move your finger horizontally or vertically first. Also try changing the PanningDeceleration to 0.0, 0.0001, and 0.01 and testing with a flick to see how far a flick will scroll the list. Finally try changing the PanningRatio to 2, 0.5, and 0 and see how the scrolling speed can be changed relative to the finger speed.

One more thing to try here is test the limits of the touch verses mouse promotion. Set the TextBox PanningMode to VerticalFirst. When you vertically scroll the TextBox, it uses the WPF Touch API and manipulations internally. When you move your finger horizontally to start text selection, it promotes the touch to a mouse event, but only if that touch is the primary touch point.

To see how this mouse promotion is limited, try this: With one finger, touch one of the scrollbar thumbs on the top ScrollViewer and move it around, but do not lift it. With a second finger on another hand, try scrolling the TextBox vertically. This should work as expected.

Now, without removing your first finger on the scrollbar, lift and place your second finger again and try to select text. This will not work. Since the text selection requires use of the mouse promotion, and the primary touch point is on the scrollbar, your second finger is ignored by the mouse system. Figure 5.3 shows the sequence of events.

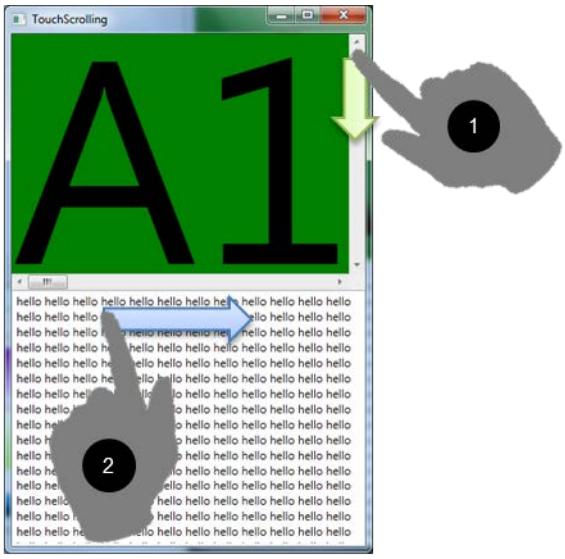


Figure 5.3 WPF does not allow interaction with multiple native controls at the same time. Only the primary touch point is promoted to a mouse event and allowed to interact with native controls. In this example, the user touched and held a scrollbar with one finger #1 and attempted to select text with a second finger #2. The first finger's touch was designated the primary touch point so the second finger is ignored and text is not selected.

This same issue exists for other combinations of native WPF controls. For example, you cannot use touch to operate a native button and a native slider at the same time. This limitation may or may not be an issue for a single user, but it certainly would limit an application designed for multiple users. Fortunately, the provided Surface SDK controls eliminate most of these issues.

You should also take note of what happens when you reach the limits of the ScrollViewer. When you try to scroll beyond the end, the ScrollViewer will report boundary feedback and the default behavior of the window is to move towards the direction you're trying to scroll. This also occurs if you flick towards an edge with sufficient velocity.

ScrollViewers in custom controls

If you are creating a custom controls that uses a ScrollViewer or SurfaceScrollViewer in the control template, users of the control can also set the panning properties on your control to customize how the embedded ScrollViewer behaves. We saw this in listing 5.2 with this line:

```
<TextBox ScrollViewer.PanningMode="VerticalFirst" [...]/>
```

This technique works with custom controls but not user controls.

If you want your control's ScrollViewer to be customizable, then do not set the PanningMode, PanningDeceleration, or PanningRatio directly on the ScrollViewer in your template. Doing that would override any attached properties. When the control loads, the ScrollViewer will look for attached properties to setup the panning behaviors.

You may want to still set sensible default values for your control. For example, the TextBox control's PanningMode defaults to VerticalFirst. The best way to implement a default value is to check and set the value in the Loaded event of your control. If you want the default to be VerticalFirst, then in the Loaded event handler check to see if PanningMode is still the default value None. If so, set it to VerticalFirst. If PanningMode was not None, then the control consumer must have set the PanningMode as an attached property.

In most cases, you will want to leave the PanningRatio to the default value of 1. One scenario where you might want it to not be 1 is if users need to scroll to a precise viewport. You could set it to a value between 0 and 1 if they don't need to scroll a lot but need to be very precise. Another option if they need to scroll a lot would be to have it default to 1 but have a mode where the PanningRatio is much smaller. This would let users scroll to the approximate position they want then use a more precise mode for specific placement.

In summary, the native ScrollViewer enhancements are good for applications where the touch experience is not a primary motivation, but a low effort addition. The native scroll bar still makes it impractical for a full multi-touch natural user interface.

At this point you should have a pretty good idea for why the native WPF controls are not practical if you want to create a great touch experience. The Surface SDK is critical to any serious touch project. Let's move on to discuss the traditional Surface SDK controls that are enhanced for touch.

5.3 Traditional surface controls

Surface SDK provides several controls that are critical if you are developing a multi-touch application. Some of these are traditional GUI controls that are simply extensions of native WPF controls that are updated and optimized for touch. Others are true multi-touch controls that are unique to the touch environment and specially designed for touch. Table 5.5 introduces the touch-enhanced traditional controls provided by Surface SDK.

Table 5.5 Surface SDK includes touch enhancements to several native WPF controls

Control	Description
SurfaceWindow	Specialized Window that sets up common defaults for a touch application and includes touch visualizations
SurfaceButton	Specialized version of native controls with touch-friendly look and feel, multi-touch activation behaviors, and touch visualizations enhancements
SurfaceCheckbox	
SurfaceRadioButton	
SurfaceSlider	
SurfaceScrollViewer	Provides touch panning with inertia as well as a touch-friendly scrollbar
SurfaceListBox	Specialized ListBox uses SurfaceScrollViewer and provides multi-touch friendly item selection
SurfaceInkCanvas	Extends the native InkCanvas to be aware of touch shapes

For the most part, the traditional Toolkit controls are drop-in replacements for the native WPF equivalents. You could do a find and replace on the names of these controls to upgrade them to the Toolkit versions and it would work, although that is a gross simplification of the effort required to make a good NUI. You have to do a little more design work than updating the versions of the controls to move from GUI to NUI.

In this section we are going to take a look at the traditional controls and learn what new functionality and behaviors they provide. It may be helpful to see the old and new controls side-by-side, so let's start by taking a look at a sample application that shows us all of the controls.

5.3.1 Native WPF and Surface control comparison

I created a sample application called SurfaceControlComparison, shown in figure 5.4, that displays each of the native and Surface SDK controls side-by-side. You can use this application to play with and learn the subtle behavior differences between the native and Surface SDK controls.

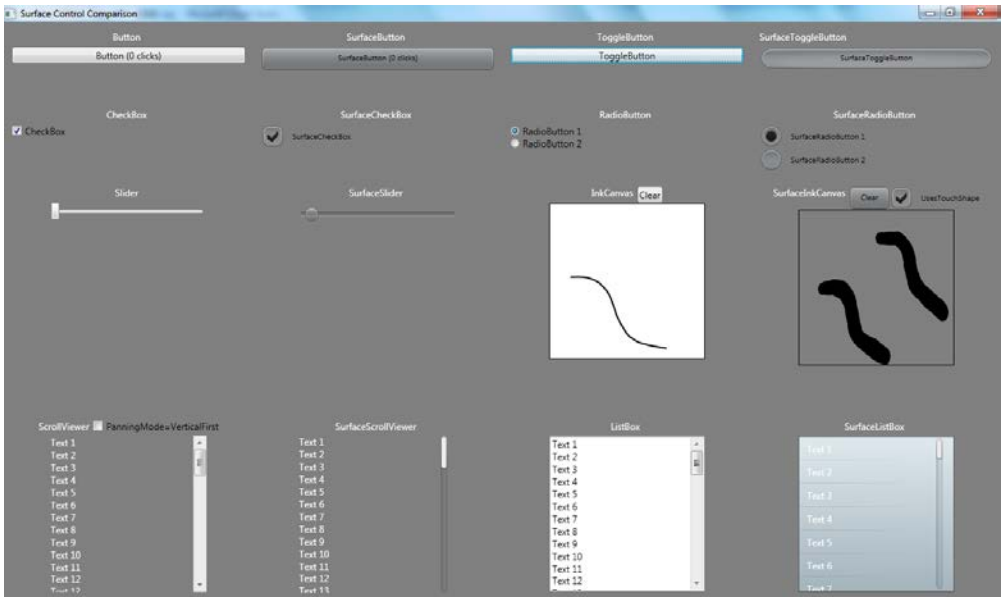


Figure 5.4 The SurfaceControlComparison sample application lets you learn the differences in look-and-feel and behavior between native WPF controls and the traditional Surface SDK controls.

All of the controls in this sample use the defaults. The most obvious difference between the native controls and the Surface controls is the visual style. The default sizes of Surface controls are larger than native controls set to make it easier for touch input. For example, the default touch target height of the SurfaceCheckBox is 26 pixels, while the CheckBox is only 13 pixels high.

The other major difference between native and Surface controls is the touch behavior. As previously discussed, each of these native controls (except the ScrollViewer's content area) only responds to the primary touch point. The Surface controls all have proper multi-touch behavior, both within a single control, such as multiple fingers touching a button, and among multiple controls, such as a finger interacting with a slider and a radio button at the same time.

Let's go through each of these controls and discuss the specific touch behaviors as well as new features available on the Surface version of the controls. Since many of these controls are derived from controls you are probably already familiar with, I will spend only as much time as needed on each control and focus only on what is different or new.

5.3.2 SurfaceButton

The SurfaceButton is a drop-in replacement for Button. The SurfaceButton captures all touches and mouse clicks (which I will refer to in general as inputs) and shows a pressed state whenever a captured input is within the bounds of the element. If all of the currently captured inputs leave the bounds of the SurfaceButton, it will show the normal unpressed state, but will return to pressed if any of the inputs move back within the bounds.

SurfaceButton still uses the Click event handler, and it only clicks if the last captured input is released while it is in the pressed state, that is, if the last input is released within the bounds of control.

This system of behavior has two implications: First, if you touch the SurfaceButton with two fingers and then release one, it will still be pressed by the other finger and not click until the second finger releases. Second, if you touch the SurfaceButton with one finger and move it outside of the control without lifting, a second finger cannot click the button. It is prevented from doing so because the first finger is still captured and could potentially return to keep the button pressed. This is an example of stolen capture.

Figure 5.5 shows SurfaceButton the normal and pressed visual states of the SurfaceButton as well as an example of how stolen capture looks.

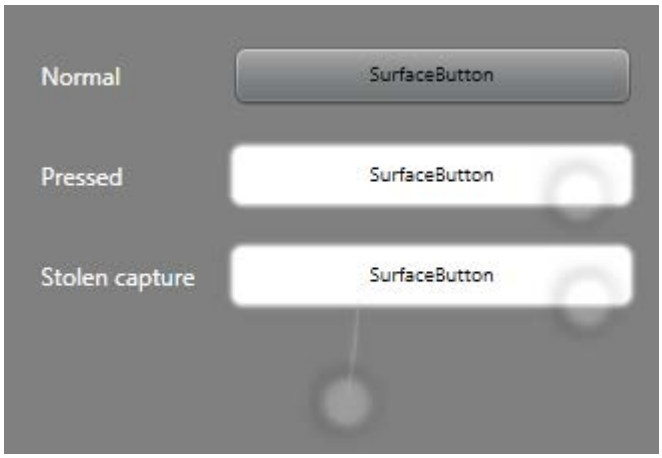


Figure 5.5 SurfaceButton has two visual states: normal and pressed. Stolen capture is also shown with one finger inside the bounds of the button and a tether drawn from the button to another finger that has moved outside the bounds of the button.

In chapter 9 we will discuss touch visualizations such as the finger auras and tether shown in figure 5.5.

5.3.3 SurfaceToggleButton

The SurfaceToggleButton is a drop-in replacement for ToggleButton. It uses the same activation behavior as SurfaceButton, except instead of clicking it will toggle between IsChecked equaling true and IsChecked equaling false. The default visual style of SurfaceToggleButton has more rounded corners than SurfaceButton.

5.3.4 SurfaceCheckBox

The SurfaceCheckBox is a drop-in replacement for CheckBox. It uses the same activation behavior as SurfaceToggleButton. SurfaceCheckBox has a large checkbox and a large touch target area that often extends above and below the text label.

5.3.5 SurfaceRadioButton

The SurfaceRadioButton is a drop-in replacement for RadioButton. Each radio button activates the similar to the SurfaceCheckBox, except the radio button stays checked with repeated activation. Within a group of SurfaceRadioButtons, only one radio button can be pressed at a time. That is, when one radio button has touches captured and causing it to be pressed, touches to other radio buttons in the same group are ignored.

5.3.6 SurfaceSlider

The SurfaceSlider is a drop-in replacement for Slider. It features visual and behavior updates that make it easier to touch and move the slider. The track starts out very thin and the thumb is round and large. Regardless of the visual size, the touch target for the thumb and track are large. When you touch the thumb or the track they grow larger. Figure 5.6 shows the various visual states and interactions of SurfaceSlider.

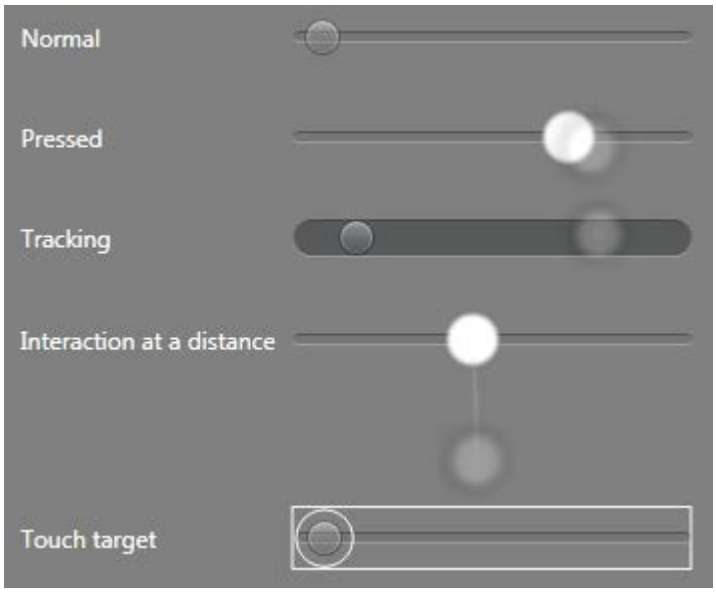


Figure 5.6 SurfaceSlider has three visual states, normal, pressed, and tracking. If you move a touch off the thumb it will show a tether to the still pressed thumb and will allow you to interact at a distance. The "touch target" slider is annotated to show the areas that respond to touch. Touching inside the white circle will activate the thumb and touching inside the white rectangle will activate the track.

The SurfaceSlider uses a different activation model than SurfaceButton. With SurfaceButton, when your touch moves outside of the control bounds the button deactivates. With SurfaceSlider, the thumb activates when you touch it and stays actives no matter where your finger goes. The thumb will follow your finger within the constraints of the track, and a tether will be drawn between the thumb and your finger when the thumb cannot go any further.

The SurfaceSlider control will also allow multiple fingers (and multiple people) to interact with it at once. If more than one touch is captured to the control, the thumb will move to the average location of all captured touches on the axis of the track. This is illustrated in figure 5.7, which also shows that the slider can be oriented vertically by setting the Orientation property to Vertical.

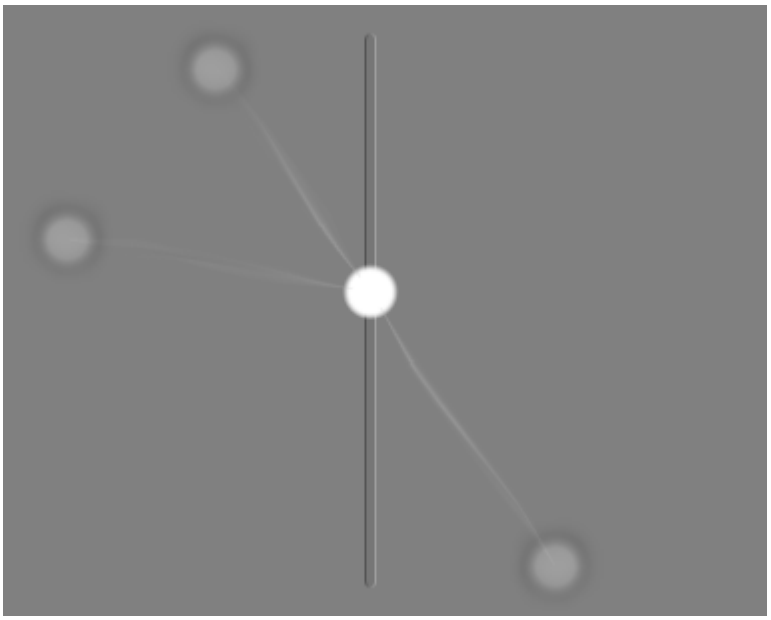


Figure 5.7 SurfaceSlider can handle multiple touches pulling on it in different directions. It will stay at the average position of all captured touches.

If you touch the track, the track will grow and the thumb will advance towards your finger, similar to a classical scrollbar. It differs though because when the thumb reaches your finger it will activate and be captured without requiring you to release and retouch the thumb.

5.3.7 SurfaceScrollViewer

SurfaceScrollViewer is an enhanced replacement for the native ScrollViewer. We discussed the couple of touch enhancements of the native ScrollViewer in section 5.1.3. SurfaceScrollViewer has all of the same features, including PanningMode, PanningDeceleration, and PanningRatio, as well as several other enhancements.

Perhaps the most important enhancement for the touch experience is the updated scrollbars. The native ScrollViewer's scrollbars only responded to the primary touch point. SurfaceScrollViewer uses SurfaceScrollBar, which are optimized for touch and operate very similar to the SurfaceSlider. The look of the SurfaceScrollBar is different from SurfaceSlider to easily distinguish between the two controls. Figure 5.8 shows the SurfaceScrollViewer and new ScrollBars in action.

Figure 5.8 SurfaceScrollViewer further enhances the touch experience by using the SurfaceScrollBar. When you touch the touch target of the track or thumb, the scroll bar will expand and allow you to scroll. When you reach the end of the scrolling area using touch panning, SurfaceScrollViewer will allow you to go slightly beyond with some elasticity. You can also keep scrolling using touch panning outside the bounds, and a tether will be displayed to highlight the distance interaction.

There is a behavior difference between SurfaceScrollViewer and native ScrollViewer when you reach the end of the scrolling area. With ScrollViewer, the scroll area remains rigid and the entire window moves. With SurfaceScrollViewer, you can actually scroll past the end of the content a little bit within the control in an elastic way. These are both valid ways to show feedback of the scrolling constraint to the user but they reflect different philosophies on interaction.

The ScrollViewer feedback depends upon the concept of a window. In many touch experiences, there are no windows and many NUI applications display full screen. Providing the feedback within the scrolling control makes more sense in these cases.

Scroll bar versus touch panning

With the scroll bar working well with touch, you may notice that there are two contradictory mechanisms of scrolling. First is the classic scrollbar scrolling, where moving you finger down visually moves the content up. Second is the new, and perhaps more natural, touch panning, where moving your finger down moves the content down.

The conflict occurs because there are two different metaphors for scrolling. The scroll bar metaphor has you controlling the virtual viewport into a larger space, similar to moving a video camera down a long document, while you are only seeing the viewport. The touch panning metaphor has you controlling the content itself, like moving a document up and down underneath a fixed camera.

It would seem that with the panning capability, we would not need the more indirect scroll bar. In many cases this is true and you would want to set the scrollbars to hidden. In some cases, though, you may still want the scrollbar. For example, if you need to show the current relative position within a large content area, the scrollbar visual takes care of that. Second, if you need to quickly jump from one place to another in a large content area, then it would be quicker to make once movement with the scrollbar than multiple flicks with touch panning.

The Surface controls up till now have all been direct replacements for the native controls. SurfaceScrollViewer adds more features and has specialized touch behavior, so there are a few properties that you can customize. Table 5.6 shows the customizable SurfaceScrollViewer properties.

Table 5.6 Use these SurfaceScrollViewer properties to customize the touch experience

Property	Type	Default value	Description
Elasticity	Vector	(0.4, 0.4)	Determines how far panning can go past the end of the content area. The Vector X and Y components represent horizontal and vertical scrolling, respectively. Values must be greater or equal to zero and represent how far past the content area the user can scroll as a fraction of the SurfaceScrollViewer's width or height.
HitTestContentWhileScrolling	bool	true	Determines whether the content area is hit test visible when the scrolling velocity is above a hard-coded threshold. Depending upon the content in the scroll viewer, you may want to set this to false to improve performance and prevent accidental activation of buttons or other content during fast scrolling operations.
IsScrolling	bool	false	A read-only property that is true if any fingers are captured for scrolling or if the scroll viewer is still moving with inertia, and false otherwise.

The SurfaceScrollViewer only captures fingers for scrolling if they are not captured by children controls and then only once they start moving. This allows you to still interact with content and controls inside the ScrollViewer.

5.3.8 SurfaceListBox

SurfaceListBox is a drop-in replacement for the native ListBox. It uses SurfaceScrollViewer in its control template, so it automatically gets all the features of the SurfaceScrollViewer, including SurfaceScrollBars and elasticity. Figure 5.9 shows the visual style and interaction of a SurfaceListBox.

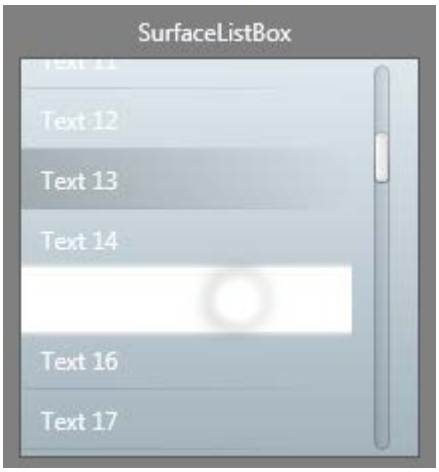


Figure 5.9 SurfaceListBox presents a list of selectable items within a SurfaceScrollViewer. In this screenshot, one item is selected and another one is pressed. In the default single-selection mode, when that finger is released it will become the new selected item.

SurfaceListBox's item container is the SurfaceListBoxItem. The native ListBoxItem only changes visually once it is selected. The SurfaceListBoxItem has a selected visual state as well as a pressed visual state. This provides more feedback to the user about what will happen when he or she touches it. The SurfaceListBoxItem exposes a bool IsPressed property, which is true if the user if currently pressing the item.

5.3.9 SurfaceInkCanvas

The SurfaceInkCanvas control replaces and enhances the native InkCanvas control for touch scenarios. InkCanvas allowed users to draw ink strokes with a mouse or stylus. SurfaceInkCanvas extends that behavior to enable inking multiple strokes simultaneously as well as touch shape inking, which draws the strokes using the touch shape that cause the stroke. This means if you use the tip of your finger you will get a thin line but if you use your whole finger or even the side of your hand you'll get a very thick line. This feature can be turned off with the UsesTouchShape property so you can still create thin lines with fat fingers.

NOTE

The UsesTouchShape feature requires the touch hardware to report the TouchPoint size. Some touchscreens do not report this information.

Figure 5.10 shows some ink strokes with and without this touch shape inking.

Figure 5.10 A SurfaceInkCanvas with two strokes. The top stroke was drawn with UsesTouchShape = true by starting the stroke with the pad of a finger and ending with just the tip. The middle and bottom strokes were drawn at the same time with UsesTouchShape = false.

Most of the other traditional Surface controls are derived from their native WPF counterpart. For example, SurfaceButton is derived from Button and even SurfaceScrollViewer is derived from ScrollViewer. This means that those controls only extend functionality, but do not take any away.

SurfaceInkCanvas, however, derives from FrameworkElement rather than InkCanvas. Internally, SurfaceInkCanvas has an InkCanvas as a child under a transparent, hit-test visible Grid blocking touch and mouse input to the InkCanvas. In essence, the SurfaceInkCanvas wraps InkCanvas and only exposes the subset of its functionality that makes sense in a touch environment.

Since InkCanvas is a unique control and many may not be familiar with it, I'm going to spend a little more time than usual explaining what SurfaceInkCanvas added and removed from InkCanvas and how to use it in general.

SURFACEINKCANVAS ADDITIONS

This control adds two features that the native InkCanvas does not have. First, it supports inking multiple strokes at the same time. Second, allows the shape of the touch to determine the shape of the ink and exposes the boolean UsesTouchShape property. This property default to true, enabling touch shape inking. Setting it to false will cause stokes to use the default ink shape.

SURFACEINKCANVAS SUBTRACTIONS

SurfaceInkCanvas does hide some functionality that does not work well in a touch environment. One thing you may notice is that the EditingMode uses the SurfaceInkEditingMode enum, and this enumeration of modes is missing some of the original InkCanvasEditingMode values. Let's look as the two enumerations:

```
public enum InkCanvasEditingMode
{
    None,
    Ink,
    GestureOnly,
    InkAndGesture,
    Select,
    EraseByPoint,
    EraseByStroke
}
```

```
public enum SurfaceInkEditingMode
{
    EraseByStroke,
    EraseByPoint,
    Ink,
    None
}
```

SurfaceInkCanvas does not support the GestureOnly, InkAndGesture, or Select modes. To understand why, we must consider the origins of InkCanvas.

The InkCanvas control and the ink and stylus infrastructure was introduced with the Microsoft Tablet PC Platform SDK 1.5 in 2003. At the time, Tablet PC applications were envisioned as GUI applications plus stylus enhancements, including handwriting recognition, ink analysis, and ink gesture recognition. Users might use regular toolbars but also draw ink in an InkCanvas. The ink analysis and gesture recognizer would have been used to tell the difference between strokes intended to be text, drawings or sketches, and strokes intended to be recognized as a gesture.

Today, most of those features are now available in the .NET Framework but the direction of interface design has evolved. Instead of gestures being constrained to a stylus interacting with an InkCanvas, NUI application designers will more likely want to integrate gestures and manipulations using touch and stylus into the core of the interface. This means the gesture recognition aspects of the InkCanvas (which frankly were very limited) are no longer needed.

The Select mode was use to enable users to draw a lasso around ink to select it then pan or resize the selected strokes. In concept, selecting and moving ink strokes would still be a useful feature in a NUI. In practice, though, the existing selection interaction used a very GUI-centric style.

First, when you activate Select mode, it puts all touches into a particular mode with no visual clues that fingers are in that mode. In a multi-touch and multi-user environment, this may cause confusion depending upon the context of the interface. Of course, this criticism could be applied to the two erase modes as well, but the erase modes do not have the same problem below.

Second, once ink is selected, it displays a traditional dotted rectangle with resize handles at the corners. These visuals, shown in figure 5.11, are very GUI-like and do not work well with touch.

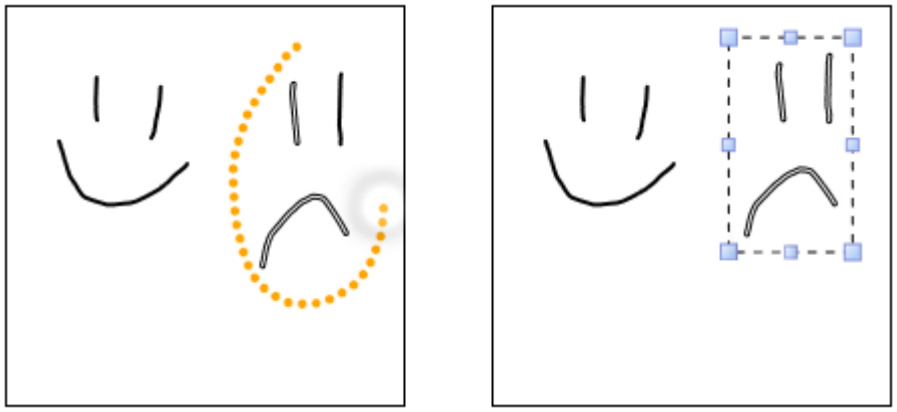


Figure 5.11 The WPF's native InkCanvas has a stroke selection mode that SurfaceInkCanvas does not implement. On the left, a finger is lassoing several ink strokes. Notice how the selected strokes visually change. On the right, the selected strokes are surrounded by a traditional GUI selection box with handles that are too reliably small to use with touch.

Since SurfaceInkCanvas does not implement selection, it also does not have several other features that the native InkCanvas has. This includes several methods that depend upon the concept of selection, including cut, copy, and paste methods.

If you had to, you could still implement a selection feature in your application. You would need to keep in mind the reasons why the original select interaction was cut and implement an interaction and visuals that work best for your users, keeping in mind the principles of NUI.

SURFACEINKCANVAS CAPABILITIES

You might wonder, after removing those features, what is left? You can still draw and erase ink, access the individual strokes, get notified when strokes are added or deleted, and set drawing properties such as color and shape. Table 5.7 summarizes the entire SurfaceInkCanvas public interface.

Table 5.7 SurfaceInkCanvas provides these public events and properties

Type	Name	Description
Event	DefaultDrawingAttributesReplaced	Occurs when the default drawing attributes change
Event	EditingModeChanged	Occurs when the editing mode is changed
Event	StrokeCollected	Occurs when a stroke is completed
Event	StrokeErased	Occurs when a stroke has been erased
Event	StrokeErasing	Occurs just before a stroke is erased and gives you a chance to cancel the erase
Event	StrokesReplaced	Occurs when a collection of strokes is replaced
Property	Brush Background	This brush is the background of the SurfaceInkCanvas
Property	DrawingAttributes DefaultDrawingAttributes	These drawing attributes are applied to new strokes

Property	SurfaceInkEditingMode EditingMode	Allows you change the current editing mode
Property	StylusShape EraserShape	Determines the shape of the object that erases strokes
Property	StrokeCollection Strokes	A collection that holds all collected strokes
Property	bool UsesTouchShape	Determines whether strokes will use the touch shape for the stroke point size

With this core functionality, the SurfaceInkCanvas control is well suited for annotating features, such as drawing ink over a photo, or for light handwriting or drawing input. If necessary, you could extend the functionality to include selection and other features, but sometimes it would be better to revert back to InkCanvas instead.

WHEN TO USE INKCANVAS INSTEAD OF SURFACEINKCANVAS

SurfaceInkCanvas is designed specifically for a touch environment and does not have all of the features of InkCanvas. In some cases, you may want to use InkCanvas instead of SurfaceInkCanvas.

The first case when InkCanvas would be preferred is if the benefits of having stroke selection, including copy and paste, outweigh the benefits of being able to draw more than stroke at the same time. This would be the case in many single-user applications on desktop or laptops. If multiple people may be using the application at the same time, this becomes a more difficult choice.

The second case when InkCanvas would be preferred is when the user needs to do handwriting input and a stylus is available. If there is no stylus, then handwriting input with fingers is equally good (or bad) with InkCanvas and SurfaceInkCanvas. Writing text with fingers is tiring and not very accurate, though, so a stylus would be preferred if one is available. I recommend the InkCanvas over SurfaceInkCanvas in the case of handwriting with a stylus for reasons that become apparent in figure 5.12.

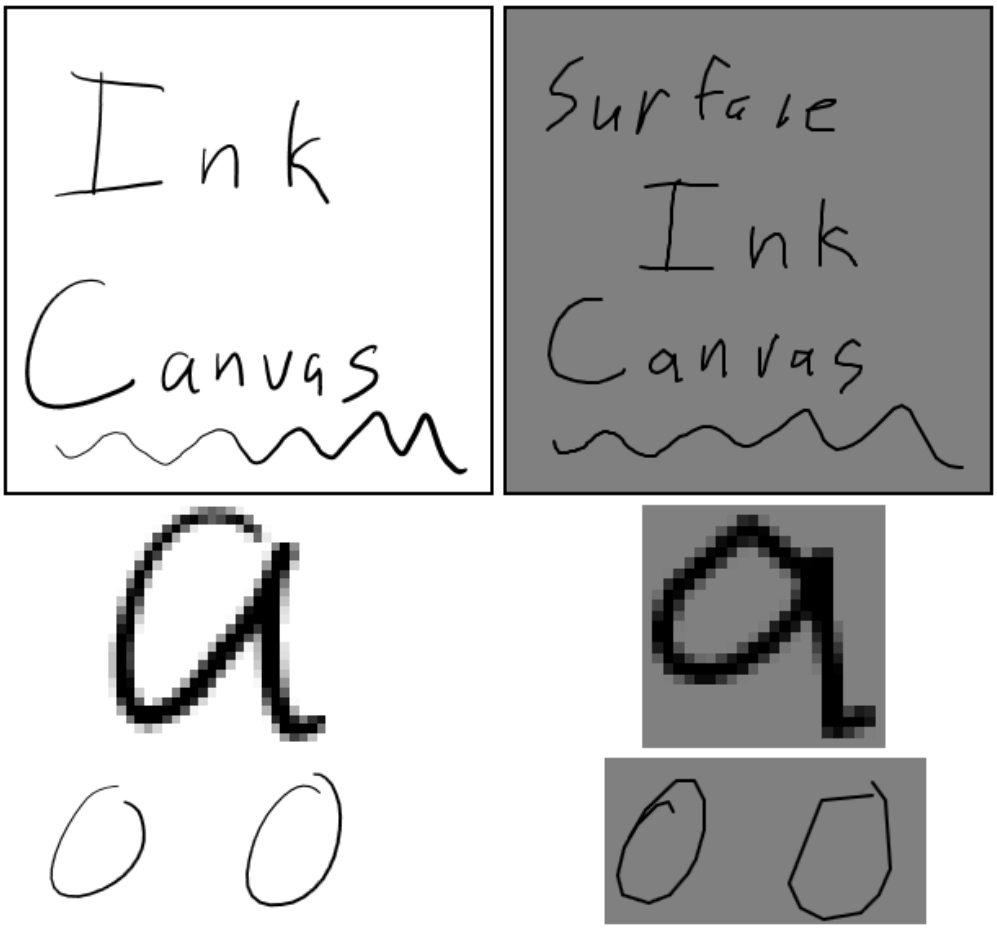


Figure 5.12 The same text was written in with a stylus at the same speed on an InkCanvas, left, and SurfaceInkCanvas, right. The InkCanvas performs significantly better with stylus input. On the top row, the text and wavy line illustrate stylus pressure sensitivity. The middle row shows the first "a" in Canvas enlarged and the bottom row shows two additional tests of a circle drawn quickly. The InkCanvas results on left are smooth and clean. You can see the line segments in the SurfaceInkCanvas results on the right.

InkCanvas handles the stylus events directly and has full access to the high-resolution stylus movements and pressure. SurfaceInkCanvas relies on mouse promotion of the stylus events. This limits the information it has and consequently the SurfaceInkCanvas ink from a stylus is not nearly as smooth or natural as the InkCanvas ink. You can see this in figure 5.12 where letters are visible as line segments rather than smooth curves. Particularly noticeable are how the beginning of strokes are truncated so much that letters are malformed, such as the "c" in Surface or "v" in Canvas, and the circles on the bottom row. I'll admit that my handwriting is not that nice, but it is not as bad as SurfaceInkCanvas makes it look.

In summary, you may want to strongly consider sticking with the native InkCanvas if you anticipate stylus and handwriting input or stroke selection to be important. In multi-user scenarios and when those factors are not as important, you can probably use SurfaceInkCanvas.

We have gone through most of the traditional Surface controls. The last one, SurfaceWindow, may not typically be thought of as a control, but is still an important part of setting up your multi-touch applications.

5.3.10 SurfaceWindow

SurfaceWindow is a drop-in replacement for the Window. It provides three main enhancements over the standard Window:

- Automatically enables touch visualizations for the entire window
- Defaults to the maximized state
- Disables pen and touch flicks

Let's discuss each of these enhancements briefly.

ENABLES TOUCH VISUALIZATIONS

SurfaceWindow includes in its control template a TouchVisualizer control. This allows touch visualizations to be registered and displayed across the entire window.

DEFAULTS TO MAXIMIZED

SurfaceWindow defaults to maximized because it is a more logical default for a touch application. If the window started normally, many users may want to move or resize it and doing those with touch is difficult. Many applications will want to use as much screen area as possible since the touch target need to be large anyway, so maximized is a logical default.

In some cases you will want to make the window full-screen. All you need to do for full-screen is to maximize the window then set the WindowStyle property on you SurfaceWindow:

```
<s:SurfaceWindow [...]
                 WindowState="Maximized"
                 WindowStyle="None">
[...]
```

This setting eliminates the titlebar and other window chrome. In some cases, you may still see a single-pixel line on the sides of the screen. This can be eliminated by setting the ResizeMode property:

```
<s:SurfaceWindow [...]
                 WindowState="Maximized"
                 WindowStyle="None"
                 ResizeMode="NoResize">
[...]
```

This guarantees a completely full-screen window. Keep in mind that you will need to provide a way for your users to exit the program. The traditional way that most users will understand immediately would be to provide a control with an "x" in the top-right corner, but you may want to do something different depending upon the specific needs of your application and users.

DISABLES FLICKS

The pen and touch flicks are a Windows feature that allows you to trigger certain navigation and editing tasks when you perform a flick gesture. When the flick gesture is recognized, an icon appears briefly. In the normal course of performing manipulations and gestures in your application, users may accidentally trigger one of these flicks and be confused. Helpfully, SurfaceWindow automatically disables these flicks for the window.

We have covered all of the traditional Surface controls provided by Surface SDK. In the next chapter we will be discussing the new, special controls that were designed specifically for touch.

5.4 Summary

In this chapter we were introduced to the Surface SDK, learned about how and why native WPF controls including ScrollViewer respond to touch, and learned about the traditional Surface SDK controls.

The Surface SDK is a critical part of WPF multi-touch development. If you need to use any traditional controls in your multi-touch application, then you'll want to use the Surface SDK versions, with the possible exception of InkCanvas.

In many applications, you'll need more than just the traditional controls. In the next chapter, we will take a look at the multi-touch controls that Surface SDK provides. Then in chapters 7 and 8 we will learn about the WPF Touch APIs that you can create your own controls.

6

Data binding with ScatterView

In chapter 5 we were introduced to Surface SDK and learned about the touch versions of some of the native WPF controls. This chapter introduces you to the Surface SDK controls specially designed for touch and goes in depth into one of these controls, the ScatterView. We cover the rest of the controls designed for touch in chapter 7. Considered together, these controls will give you a good starting point for a lot of simple multi-touch interactions.

> **NOTE**
>
> This chapter is written against a pre-release version of Surface SDK 2.0. The final version of the SDK may include changes and this chapter will be updated for the final release. Please also be aware that until Surface SDK 2.0 is released publicly, some content is still covered by NDA and may be hidden from MEAPs. The final release of this book will obviously include all of the content.

While these controls are discussed in the MSDN documentation, the sample code provided in the documentation is sometimes very simple and does not reveal subtle behaviors in actual use. Rather than just reviewing the basics about how these multi-touch controls work, I am going to discuss them in terms of a real-world data-binding scenario.

In this chapter, we are going to build out an application that shows you how to use the ScatterView control. Because we will be walking through a lot of code and have plenty of screenshots along the way, this chapter will be a little longer than normal. It will be worth it, though, because this chapter will teach you everything you need to know about using the ScatterView control in practical applications.

To begin, we're going to discuss the new multi-touch Surface SDK controls at a high level and introduce some common code we'll be reusing throughout the chapter in our sample application. Next we will spend some time with the ScatterView, learning how to data-bind with content and making the content behave how we want. We will also spend some time discussing how to customize the look and feel of ScatterViewItems.

6.1 Controls designed for touch

In addition to the touch optimized versions of native WPF controls we covered in chapter 5, Surface SDK provides five controls that were designed specifically for touch scenarios plus one control designed for Surface hardware. While you could use some of the features of these controls with a mouse, they really require multi-touch and live entirely in the NUI world.

This section will provide an overview of the new Surface SDK controls, shown in table 6.1. We will be discussing the ScatterView control in detail in this chapter and the rest of these multi-touch controls in chapter 7.

Table 6.1 The Surface SDK provides six controls designed specifically for touch

Control	Category	Description
ScatterView	Designed for touch	Enables child elements to pan, rotate, and scale using touch, enabling content to be arranged in a free-form manner
LibraryStack	Designed for touch	Displays children stacked upon each other and enables users to sort through them
LibraryBar	Designed for touch	Displays children arranged in a wide grid organized into groups
LibraryContainer	Designed for touch	Allows users to switch between a LibraryStack mode and LibraryBar mode for content.
ElementMenu	Designed for touch	Allows exploration and activation of a hierarchy of menu commands
TagVisualizer	Designed for Surface	Displays and tracks visualizations at recognized tags

Within the OCGM model, all of these controls can be considered containers. They each arrange content in different ways. For example, the ScatterView is appropriate for free-form and loosely organized or unrelated content, while the LibraryStack and LibraryBar are appropriate for logically grouped content. Figure 6.1 shows a preview of what these controls look like.

Figure 6.1 The ScatterView control is not directly visible but spans the entire image and enables free-form layout of the content #1 and controls. The circular LibraryStack #2 presents content in a stack while the longer LibraryBar #3 presents content in a grid format. LibraryContainer #4 can switch between the stack and bar views dynamically.

LibraryStack and LibraryBar are really designed to work together as two different ways to visualize objects. They are designed to make it easy to drag-and-drop content from one to the other. The LibraryContainer is a hybrid of LibraryStack and LibraryBar and allows users to dynamically switch between the views.

Additional paragraph about ElementMenu and TagVisualizer excluded from the MEAP due to NDA restrictions.

In this chapter and the next chapter we are going to learn how to use each of these controls. For this chapter we will focus on the ScatterView control. Since these controls are used to show collections of items, I'll show examples using data binding scenarios that will help you get started using these controls in real-world applications.

In order to help with the data binding examples, I have prepared some code that we will build upon. Let's take a look at the basic code and then we will see how to incorporate the ScatterView control into the project.

6.1.1 ExploringTouchControls

As we explore ScatterView and the library controls, we're going to reuse a bit of code. I'll introduce that code here and will refer to it in each of the following samples. The overall sample that we'll be using is the ExploringTouchControls sample application.

> **NOTE**
>
> In the sample code download, I have provided a folder called ExploringTouchControls.Start, which represents the initial state of the project, and ExploringTouchControls.Complete, which represents the final state of the projects. If you wish to follow along with the coding, then use the .Start solution. Otherwise you can just read and then look at the final product with the .Complete solution.

Within the ExploringTouchControls solution, there is a project called DataControls. This contains a data model class, a user control that displays the data model, and a factory class that creates the data models. For demonstration purposes, I have chosen the data model to represent U.S. state flags and have included a directory with the images of the state flags. Table 6.2 summarizes the classes in this project.

Table 6.2 The DataControls project includes the following classes

Class	Description
StateModel	Represents the data for each state. Exposes StateName, FirstLetter, and FlagURI properties.
StateModelFactory	Creates and populates a List<StateModel>. Creates one StateModel for each PNG file in the flags directory.
StateView	A user control that visualizes the StateModel. Displays the state name and an image of the flag below it.

The StateView class is a user control that is designed to use data binding to display a StateModel. The StateModel itself does not have any awareness of visuals. It is basically holds the pure data and exposes the data as properties.

> **MVVM**
>
> Readers familiar with the Model-View-ViewModel pattern, or MVVM, will understand the naming of these classes. The MVVM pattern provides separation of data from presentation in a loosely-coupled architecture. While I will be using MVVM in sample applications and I encourage readers to use it as well, teaching MVVM is outside the scope of this book.
>
> In a complete MVVM implementation I would have a StateViewModel to mediate between the StateView and StateModel, but this example is simple enough that that would just be an unneeded layer.
>
> In later chapters, I will show more complete examples of implementing MVVM in a NUI.

StateView data binds to those properties, so it is important for the StateModel to implement the INotifyPropertyChanged interface and call the PropertyChanged event whenever a property is changed. Otherwise the WPF data binding system will not know to update the visuals that depend upon those properties. Figure 6.2 illustrates this data binding process.

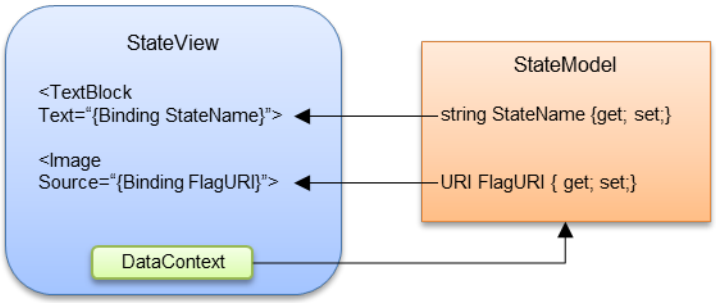


Figure 6.2 Controls in the StateView using data binding to get values. The StateView's DataContext will be set to a StateModel instance. The data binding system searches up the visual tree for a DataContext, then gets values for the matching property names. If the StateModel values change, they will automatically be updated in the StateView since StateModel implements INotifyPropertyChanged.

If you wanted, you could create a StateView and StateModel instance and manually set the DataContext. In most applications we will want to take advantage of ItemsControls such as ListBox to setup the DataContext for us.

ScatterView, LibraryStack, LibraryBar, and LibraryContainer are also ItemsControls and we will be demonstrating how to use them by data binding to StateView and StateModel.

Now that we have covered the background information about the Surface SDK controls designed for touch and the starting point for our data binding sample applications, let's get the know the ScatterView control and then incorporate it into the sample application.

6.2 Getting to know ScatterView

We already saw a very simple example of the ScatterView control in section 5.1.1 with the Hello, Surface SDK sample. In that sample, we put our content (a rectangle) inside of a ScatterViewItem, which itself was inside of a ScatterView. That sample showed the most basic implementation of ScatterView -- adding hard-coded visual elements directly to the ScatterView.

In reality, we didn't need to specify the ScatterViewItem control. Just like a ListBox and the optional ListBoxItem, the ScatterView will automatically wrap each item in a ScatterViewItem. The ScatterViewItem is important because it tells the ScatterView how to position the content.

> **NOTE**
>
> By convention, in code ScatterViewItem references are often named SVI. I will also use this abbreviation in the text at times to avoid writing out and repeating the full ScatterViewItem too many times.

While it is easy to directly add visuals to a ScatterView, it is very limiting. In most real-world applications using ScatterView you will need to create a data-driven interface. We can use ItemTemplates and data binding to achieve this.

Before we dive into details of the ScatterView and ScatterViewItem classes, let's start with a quick example of how you can drive a ScatterView with a data model.

6.2.1 Data-driven ScatterView

Here is a typical scenario for many multi-touch NUI applications: You must load data or content from some data source then display it in an interactive way. In principle, this applies to many different types of applications, not just multi-touch with ScatterViews. In order to get the ScatterViewItems to display correctly, though, we sometimes need to take a few extra steps, as we will see.

Within the ExploringTouchControls solution, add a new project called ExploringScatterView. Make sure you use the "WPF Application (Surface)" template in the Surface | Windows Touch template folder.

We'll be using StateModel, StateModelFactory, and StateView, so add a reference to the DataControls project. Before we attempt the ScatterView, let's make sure a basic data binding scenario works with our data source. We can easily test this with SurfaceListBox. Figure 6.3 shows what we are going to do.

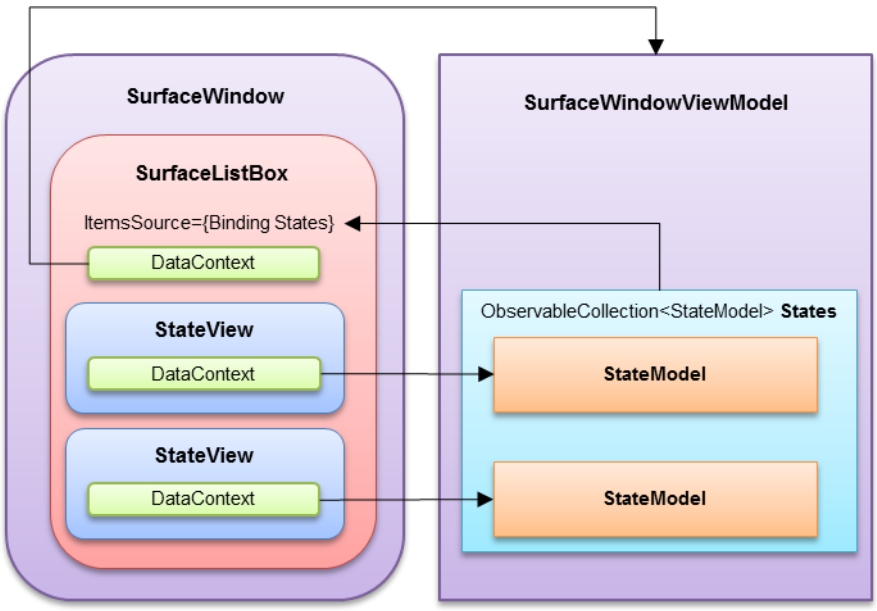


Figure 6.3 The SurfaceListBox displays StateViews data bound to StateModels. The SurfaceWindowViewModel class has an ObservableCollection of StateModels called States. The SurfaceListBox's DataContext is set to a SurfaceWindowViewModel instance so that the ItemsSource can bind to States. For each StateModel in States, a StateView is created and the DataContext is automatically set to the matching StateModel.

We are going to create a SurfaceWindowViewModel that exposes a collection of StateModels and then set the SurfaceListBox ItemsSource to bind to that collection. We will also set the SurfaceListBox ItemTemplate to the StateView. WPF takes care of the rest.

Add a new class called SurfaceWindowViewModel to the ExploringScatterView project and add these using statements at the top of SurfaceWindowViewModel.cs:

```
using DataControls;
using System.Collections.ObjectModel;
```

Next, update the SurfaceWindowViewModel class to look like listing 6.1. This will create the States collection and initialize the data using the StateModelFactory.

Listing 6.1 Add StateModel collection property and initialization logic

```
public class SurfaceWindowViewModel : INotifyPropertyChanged
{
    #region Properties

    ObservableCollection<StateModel> _states =
        new ObservableCollection<StateModel>();
    public ObservableCollection<StateModel> States              #1
    {
        get
        {
            return _states;
        }
    }

    #endregion

    #region Constructors

    public SurfaceWindowViewModel()
```

```
        {
            InitData();
        }

        #endregion

        #region Private Methods

        void InitData()
        {
            List<StateModel> models = StateModelFactory.GenerateStates();   #2

            models.ForEach(States.Add);                                     #3
        }

        #endregion
    }
```

#1 Bound to ScatterView ItemsSource
#2 Loads States
#3 Add states to bound collection

In this code, we create an ObservableCollection of StateModels #1. ObservableCollection internally implements INotifyCollectionChanged and notifies the data binding system if items are added, removed, replaced, or reordered, while a regular List does not. We also generate all of the StateModels #2 and add them to the States collection #3.

BINDING A SURFACELISTBOX

Next we need to create some visuals that make use of the exposed data. Remember that we are initially testing the data with a SurfaceListBox, and then we will change it to a ScatterView.

Open SurfaceWindow1.xaml and add these namespaces to the root SurfaceWindow element:

```
xmlns:data="clr-namespace:DataControls;assembly=DataControls"
xmlns:local="clr-namespace:ExploringScatterView"
```

Now replace the contents of the SurfaceWindow element with listing 6.2.

Listing 6.2 Initial XAML to data bind a ListBox to StateModels

```
<s:SurfaceWindow.DataContext>
    <local:SurfaceWindowViewModel />                    #1
</s:SurfaceWindow.DataContext>
<s:SurfaceWindow.Resources>
    <DataTemplate x:Key="StateTemplate">                #2
        <data:StateView />                              #2
    </DataTemplate>                                     #2
</s:SurfaceWindow.Resources>
<Grid Background="Gray"
      Name="LayoutRoot">
    <s:SurfaceListBox
         ItemsSource="{Binding States}"                  #3
         ItemTemplate="{StaticResource StateTemplate}"   #3
         HorizontalAlignment="Center"
         VerticalAlignment="Center"
         Margin="10"
         Width="400"
         Height="500" />
</Grid>
```

#1 Creates ViewModel; sets to DataContext
#2 Use StateView control as DataTemplate
#3 Set up ItemsControl data binding

Here's is what is going on in the XAML in listing 6.2. First, we set the entire window's DataContext to a SurfaceWindowViewModel instance #1, which is actually created by XAML. Next, we defined a DataTemplate #2 that just contains the StateView control. In the SurfaceListBox, we bind the ItemsSource to the States property and specify the StateTemplate resource as the ItemTemplate #3. What this does is tell WPF for each object in the States collection, create a StateTemplate data template

and set the DataContext of the template instance to the object from States. This allows the data binding of the StateView control to work like figure 6.2.

If you run the application now, you will see the SurfaceListBox displaying our StateModel data, shown in figure 6.4.

Figure 6.4 The SurfaceListBox displaying StateModel data from the States collection using the StateView as the ItemTemplate.

Now that we have seen the data binding in action (and hopefully refreshed our memories about how data binding works), let's change the SurfaceListBox to a ScatterView.

BINDING A SCATTERVIEW

In SurfaceWindow1.xaml, delete the SurfaceListBox and replace it with this code:

```
<s:ScatterView Name="scatter"
               ItemsSource="{Binding States}"
               ItemTemplate="{StaticResource StateTemplate}" />
```

Now we can get our first glimpse of how the ScatterView looks using data binding, seen in figure 6.5. Unfortunately, it is not quite what we expected.

Figure 6.5 The ExploringScatterView sample in our first attempt at a data-bound ScatterView. The ScatterViewItem do not have a good default size and content clips instead of scaling. Here I am expanding the Virginia object with two fingers, indicated with the circular touch visualizations.

While we do have all of the data and we can move items, the ScatterViewItem's default size and aspect ratio does not match the content. In addition, we might expect the content to scale with the size of the SVI but in this case it just clips as the viewport gets larger or smaller.

The reason why this doesn't "just work" is because the SVI has to make some assumptions about the size and nature of its content for it to scale. The default behavior, as we saw, is essentially the lowest common denominator of all assumptions and gives you the most flexibility.

Suppose that the ScatterViewItem automatically scaled the content. This would make things a tiny bit easier for us right now, but there are valid scenarios when you do not want the content to just scale. You might have content, such as text, that uses WPF's powerful layout engine to reflow itself. Working from an auto-scaling SVI back to a reflowing SVI would be a large challenge.

ScatterViewItem cannot make any assumptions about the default size and aspect ratio either, for the same reasons. In our example, the content has fixed default dimensions that the SVI could conceivably pick up on, but content that has a dynamic layout may have many valid aspect ratios and it would be impossible to pick one.

With those explanations out of the way, let's see how we can customize the behavior to do exactly what we want.

SCALING THE CONTENT

Here is where we need to do a few extra things with the ScatterView to make it behave properly. First, let's fix the scaling issue. This turns out to be really simple.

Open StateView.xaml. Notice that we have a Grid named LayoutRoot that contains a Border. The Border has a StackPanel with a TextBlock for the state name and Image for the state flag. What we need to do is wrap the StackPanel with a Viewbox. Listing 6.3 shows the modified XAML.

Listing 6.3 StateView.xaml needs a Viewbox to enable ScatterView scaling

```
<Grid Name="LayoutRoot">
    <Border Name="border"
            BorderBrush="Black"
            Background="White"
            BorderThickness="1">
```

```
        <Viewbox Name="viewbox">                                    #A
            <StackPanel Margin="5">
                <TextBlock Text="{Binding StateName}"
                           HorizontalAlignment="Center"
                           FontSize="24"
                           Margin="5" />
                <Image Source="{Binding FlagURI}"
                       Margin="5"
                       Width="300" />
            </StackPanel>
        </Viewbox>                                                  #A
    </Border>
</Grid>
```

#A Wrap StackPanel with Viewbox

If you run the application now, then you will get a result like figure 6.6.

Figure 6.6 With the Viewbox wrapping the content, it scales with the ScatterViewItem.

Now the content scales with the ScatterViewItem. The ViewBox lets you arrange content with hard-coded values but scales the content to uniformly fill the container. The only remaining problem is the default size of the ScatterViewItem.

SETTING THE SCATTERVIEWITEM SIZE

Once we set the size, it will maintain the aspect ratio. There are several approaches to setting the ScatterViewItem's size.

We saw one approach in the Hello, Surface SDK sample. There we explicitly created the ScatterViewItem in XAML and set the Width and Height. The obvious limitation of this is that it isn't scalable and does not really support data binding like we're trying to do in the current sample.

Another approach would be to create ScatterViewItems in code, set the size and content, then add it to the ScatterView or an ObservableList the ScatterView is bound to. This would certainly work, but it is not a very effective pattern. You would lose a lot of existing functionality like data templates and end up with a fragile, hard-to-maintain codebase.

One technique that you might think would work is changing the ItemTemplate. You could change the data template to this include a SVI and bind the Width and Height properties to your model, assuming you add those properties to the model:

```
<DataTemplate x:Key="StateTemplate">
    <s:ScatterViewItem Width="{Binding Width}"
                       Height="{Binding Height}">
```

```
            <data:StateView />
        </s:ScatterViewItem>
    </DataTemplate>
```

This actually would not work. You would end up with your StateView within a ScatterViewItem you provide, but that is wrapped in another SVI created by the ScatterView. The inner SVI captures the touch input making it impossible to move the outer SVI.

Double ScatterViewItems

For the curious reader, here is why you get the redundant second ScatterViewItem.

The ScatterView takes the direct children or data-bound items and wraps them in the appropriate container, specifically a ScatterViewItem, if they are not already ScatterViewItems. When the ScatterView prepares the item, such as the StateModel, for a container, it first checks in the IsItemItsOwnContainerOverride method whether the item is already an SVI. If it is, then it leaves it alone. If not, then it wraps it in a new SVI.

Since the ScatterView (and any other ItemsControl) checks the data-bound StateModel, not the DataTemplate, it doesn't realize that you're already specified a SVI. It wraps the StateModel in an ScatterViewItem and separately WPF applies your DataTemplate, which contains the second (inner) SVI.

The create ScatterViewItem in code approach suggested above does not run into this problem because we are adding SVIs directly as children or data-bound items.

One final approach to this issue, which will work, is walking the visual tree from the StateView object to find its parent ScatterViewItem and setting values.

To implement this, we need to modify StateView.xaml.cs. I have already provided this code already in StateView.xaml.cs, so all you need to do it uncomment the call to ConfigureSVI() in the control_Loaded method, which is called when the control has completely loaded, appropriately enough. This event handler is specified in StateView.xaml:

```
<UserControl x:Class="DataControls.StateView"
                    [...]
                    Loaded="control_Loaded">
```

Listing 6.4 shows the code that we just enabled.

Listing 6.4 Setting the SVI size when StateView is first loaded

```
private void control_Loaded(object sender, RoutedEventArgs e)
{
    ConfigureSVI();
}

void ConfigureSVI()
{
    ScatterView sv =
        FindVisualParent<ScatterView>(this);              #A

    if (sv == null)
        return;

    ScatterViewItem svi =
        sv.ItemContainerGenerator.ContainerFromItem(      #B
        this.DataContext) as ScatterViewItem;             #B

    if (svi == null)
        return;

    double aspectRatio = this.viewbox.ActualWidth /
                         this.viewbox.ActualHeight;


    svi.Height = 150;                                       #C
    svi.Width = svi.Height * aspectRatio;                   #C
```

```
}

static T FindVisualParent<T>(DependencyObject obj) where T:UIElement
{
    if (obj is T)
    {
        return obj as T;
    }

    DependencyObject parent = VisualTreeHelper.GetParent(obj);
    if (parent == null)
        return null;
    return FindVisualParent<T>(parent);                    #D
}
```

#A Find ScatterView in visual tree
#B Get SVI with same DataContext
#C Set size using correct aspect ratio
#D Recursively search up visual tree

When the control is first loaded, this code finds the parent ScatterView in the visual tree and then uses the ScatterView's ItemContainerGenerator to find the SVI with the same DataContext as this StateView. If it finds the SVI, it sets the width and height using the same aspect ratio as the content. Notice that we calculate the aspectRatio using the ActualWidth and ActualHeight of the viewbox. We cannot use the LayoutRoot grid or the control itself because those are already using the full dimensions of the SVI at this point.

Figure 6.7 shows the fully working, data-driven ScatterView.

Figure 6.7 With the Viewbox scaling and appropriate ScatterViewItem sizes, content now behaves as we would expect.

While it may not be best practice to walk the visual tree, in some cases it is justified. This code is written to be robust in a variety of situations. If it cannot find the ScatterView, then it returns without doing anything. If it does find it, but it can't find a SVI using ItemContainerGenerator and the StateView's DataContext, then it returns without doing anything.

We use this technique rather than just searching the visual tree for any ScatterViewItem because if this StateView could be used within some other control that happens to be in a SVI. For example, suppose we put the SurfaceListBox with all the StateViews inside of a ScatterViewItem. We wouldn't want all the

StateViews to change the SurfaceListBox's SVI. This code will only find and change a SVI that matches the StateView's DataContext.

We are only setting the size here, but there are several other behaviors of the ScatterViewItem that we could customize. The flexibility of this approach is that these behaviors could be different for each item and could in fact be data-driven as well. For example, we could set the Center and Orientation of the SVIs based upon saved positions stored in a data store.

Let's take a look at the ScatterView and ScatterViewItem classes to see what else we can do.

6.2.2 ScatterView and ScatterViewItem classes

The ScatterView control fills all available space and provides the context for ScatterViewItems. In most applications, you don't actually see the ScatterView, unless you set the background color. The ScatterViewItems wrap all of the content and provide access to most of the functionality to customizing the behavior of the content.

ScatterViewItems can be manipulated freely within the bounds of the ScatterView. In normal circumstances, the user cannot push or throw a ScatterViewItem outside of the ScatterView. SVIs will automatically stop and bounce back enough to make sure the user can still touch and manipulate the item. The logic is smart enough to take into account orientation and size so SVIs don't get stuck beyond the edges.

In most cases you will want to place the ScatterView so it takes up the entire window. If you needed to you could configure it to constrain SVIs to a smaller space, through setting the ScatterView's Width and Height or arranging it within other panels.

Aside from defining the bounds of the SVI movement and providing two methods, most of the class members that we are about to discuss belong to the ScatterViewItem class. These properties, methods, and events of the ScatterView and ScatterViewItem classes can be grouped into three different categories: positioning, manipulation, and activation. Instead of going through the class members in order, I will talk about each category one at a time.

In the example in the previous section we were discussing initializing the SVI size, position, and orientation, so let's start with positioning category.

SCATTERVIEWITEM POSITIONING

The positioning category includes the properties and methods that handle where a ScatterViewItem is. Table 6.3 shows the positioning properties.

Table 6.3 ScatterViewItem expose these properties to control positioning

Name	Type	Default	Description
ActualCenter	Point	random	Gets the rendered position of the center of the SVI
ActualOrientation	double	random	Gets the rendered orientation of the SVI
CanMove	bool	true	Whether the SVI can move
CanRotate	bool	true	Whether the SVI can rotate
CanScale	bool	true	Whether the SVI can scale
Center	Point	random	The center position of the SVI
Orientation	double	random	The orientation angle of the SVI
ZIndex	int	0	Determines the relative Z order among other SVIs in a ScatterView. The higher the number the closer to the top.

The properties you will use most often are Center and Orientation. These properties default to a random value that puts the ScatterViewItem within the bounds of the ScatterView. The ActualCenter and ActualOrientation properties contain the last rendered position of the ScatterViewItem. If you are

programmatically changing the position or orientation of a ScatterViewItem, you might need the ActualCenter or ActualOrientation to compare the last position with the new position that has not yet been rendered.

You may note that there is no Scale property. ScatterViewItem actually updates the Width and Height when you scale it. Since Width and Height are actual properties of FrameworkElement, they were not included in table 6.4.

When you initialize a ScatterViewItem, you'll want to set the Center, Orientation, Width, and Height. You may also want to set CanMove, CanRotate, and CanScale at the same time. These default to true, but if you set them to false you can prevent users from manipulating the Center, Orientation, and Width and Height, respectively. If you are creating an application for a vertical form factor, it may not make sense for users to rotate content. In that case you would want to initialize Orientation to 0 and set CanRotate to false.

The ZIndex property controls the relative visual order of overlapping ScatterViewItems. This can be helpful if you want to save or set the Z-order of SVIs. One of the two methods in the positioning category is also helpful for changing the ZIndex. Table 6.4 lists ScatterViewItem's positioning methods.

Table 6.4 ScatterViewItem provides these methods to that affect positioning

Method	Description
void BringIntoBounds()	If the SVI is outside the ScatterView bounds, this method moves it just enough so the user can see and manipulate it.
void SetRelativeZIndex(RelativeScatterViewZIndex relativeIndex)	Moves the SVI to the specified relative position

Even though you can only manipulate a SVI within the ScatterView bounds, you can programmatically set the location to be anywhere using the Center property, even outside the ScatterView bounds. If a ScatterViewItem is positioned outside of the ScatterView, you can call BringIntoBounds() on an SVI to cause it to jump inside the bounds just enough so the user can touch and manipulate it again. This is an instant transition, though, so you may want to consider animating the Center property instead of calling BringIntoBounds().

You can call SetRelativeZIndex() on a ScatterViewItem to easily change the Z-order without messing with the specific ZIndex values. This method takes the enumeration explained in table 6.5 as a parameter.

Table 6.5 The RelativeScatterViewZIndex enumeration is used with SetRelativeZIndex()

Value	Description
Topmost	Moves the SVI above all other SVIs, included activated SVIs
AboveInactiveItems	Moves the SVI above all inactive SVIs, but stays below the activated SVIs
Bottommost	Moves the SVI below all SVIs

You might use SetRelativeZIndex() if you want to bring certain SVIs to the top to highlight them to the user. This would make sense when adding new content to existing ScatterView content, for example. The following lines would make this happen:

```
ScatterViewItem svi = GetSVIReference();
svi.SetRelativeZIndex(RelativeScatterViewZIndex.Topmost);
```

So far the class members we covered only deal with the positioning of a ScatterViewItem. You can also change and get information about how the SVI is manipulated.

SCATTERVIEWITEM MANIPULATION

When the user touches a ScatterViewItem, the touch is captured and starts manipulating the SVI through panning, rotation, and scaling. If the user lifts all fingers from a SVI while it was still panning or rotating,

it will experience inertia. This means that it will continue moving but quickly slow to a stop as if there was friction.

NOTE

We will cover the manipulation API completely in chapter 9, so in this section I will not go into the details about the ScatterViewItem manipulations.

Table 6.6 shows three ScatterViewItem properties that can customize the manipulation and inertia behavior.

Table 6.6 ScatterViewItem exposes these movement properties

Name	Type	Default	Description
AngularDeceleration	double	0.00027	The angular deceleration when rotating under inertia
Deceleration	double	0.001536	The linear deceleration rate when moving under inertia
SingleInputRotationMode	SingleInputRotationMode	Disabled	Determines whether the SVI will rotate with only a single input

The AngularDeceleration and Deceleration determines how quickly a SVI slows down while moving under inertia. These are decent values and normally do not need to be changed unless you need a ScatterViewItem to be much easier or harder to flick across the screen.

The SingleInputRotationMode property determines whether a SVI can be rotated with a single finger. The SingleInputRotationMode enumeration has three possible values: Default, Disabled, and ProportionalToDistanceFromCenter. Default is the same as Disabled and means that no matter where you touch the SVI, a single finger does not cause rotation. The ProportionalToDistanceFromCenter option allows the SVI to pivot around a single moving finger.

To understand how the ProportionalToDistanceFromCenter mode works, imagine you are sitting at a table across from someone and want to rotate a piece of paper on the table to face them. One way you could do this is to loosely touch a corner of the paper and push it so that the paper rotates around its center. On the other hand, if you touch near the center and push the paper, it will just pan and very little rotation will occur.

The ScatterViewItem also exposes several events that notify you as a ScatterViewItem is being manipulated. Table 6.7 shows these events.

Table 6.7 ScatterViewItem manipulation events

Event name	Description
ContainerManipulationStarted	Fires when the user first touches the SVI
ContainerManipulationDelta	Fires when the user moves any touches captured to the SVI
ContainerManipulationCompleted	Fires when the last touch is released from the SVI

These events have similar EventArgs to their non-container versions, to be covered in chapter 9.

Finally, there are two ScatterViewItem methods that affect manipulation. The first is CancelManipulation(). This method stops the current SVI manipulation or inertia, even if fingers are still interacting with it. It does not affect the activation so the SVI may still be raised up if one or more fingers are still captured. New fingers can start interacting with the SVI after the CancelManipulation() call, but existing captured fingers will be ignored until they are lifted.

The second method here is BeginDragDrop(object data). This method is a shortcut for starting a drag-and-drop operation with the specified ScatterViewItem. Since we have not covered the drag-and-drop framework yet, I will wait until chapter 9 to discuss this method.

The final category of class members is activation.

SCATTERVIEWITEM ACTIVATION

I have mentioned activating the ScatterViewItem a few times so far, and there are a couple properties, events, and methods that deal directly with activation. Let me explain what this means.

When you touch a ScatterViewItem it becomes activated. Figure 6.8 shows the default visual activation states of ScatterViewItem.

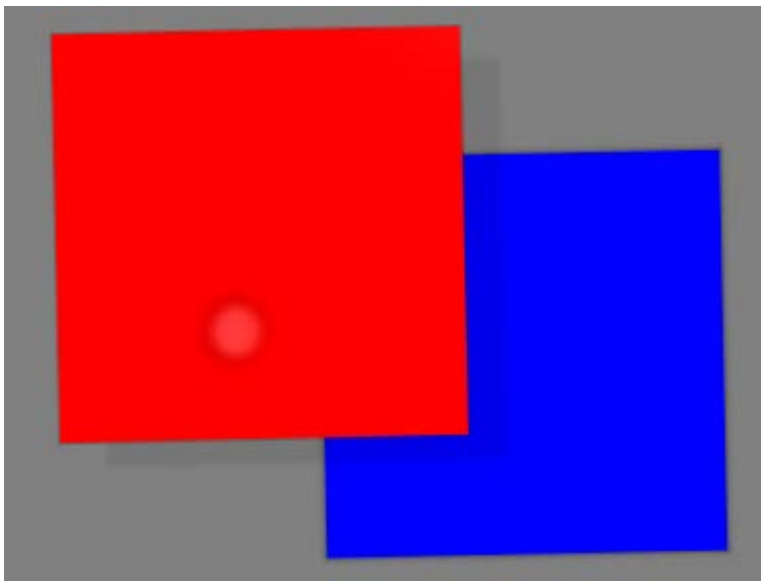


Figure 6.8 The ScatterViewItem on the right is deactivated. The ScatterViewItem on the left is activated by a touch, causing it to slightly increase in size and show a shadow. A visual sheen also animates across the content, but that effect is not shown here.

What happens when it is activated depends upon the ShowsActivationEffects and IsTopmostOnActivation properties, which default to true, and ContainerStaysActive, which defaults to false. Table 6.8 introduces the activation properties.

Table 6.8 ScatterViewItem exposes these properties to control activation behavior

Name	Type	Default	Description
ContainerStaysActive	bool	false	Whether the SVI remains active when no longer receiving input
IsContainerActive	bool	false	Whether the SVI is activated
IsTopmostOnActivation	bool	true	Whether the SVI becomes topmost on activation
ShadowVector	Vector	(0, 0)	Gets the relative position of the shadow
ShowsActivationEffects	bool	true	Whether the SVI shows a visual response to activation

If ShowsActivationEffects is true, then the SVI and its content will appear to lift from the screen. This illusion, seen in figure 6.8, is created by scaling up the content, animating a light sheen effect across the

content, and animating a shadow. When the SVI is deactivated, these animations reverse to their default state unless ContainerStaysActive is true, in which case the SVI stays visually activated until it is deactivated programmatically.

You can check the IsContainerActive property to see whether the SVI is currently active and you can set it to change the status as well. When you set `IsContainerActive = true`, it will stay active until you set `IsContainerActive = false` or the user touches and releases it.

If IsTopmostOnActivation is true, then when the SVI is activated the ZIndex is changed to bring it on the top of all other active or inactive SVIs. This order persists even after deactivation. If IsTopmostOnActivation is false, then as long as ShowsActivationEffects is true the SVI will still visually animate, but it may still be behind other deactivated SVIs.

ScatterViewItem activation effects and usability

The ScatterViewItem activation effects visuals are subtle but important for usability. The instant feedback gives the user confidence that the computer is responsive. The visuals also help the user understand what they can do with the content they are touching.

The visual of content lifting to meet your touch is an extension of the object metaphor, discussed in chapter 2. The content smoothly transitions, as a real-life object would, from laying on the display surface to hovering above it as if you had picked up the object. Of course, objects in the real world don't hover just because you point or touch them, but that is part of the magic of the interaction.

There are many possible implementation of this interaction. For example, if you can detect pressure with your touchscreen, or reliably emulate it with the size of a touch point, you could change the ZIndex based upon pressure. Alternatively, if you just want a flat visual style, you could animate the border or color of an item instead of using scale and shadow.

Of course, those alternatives are not a part of the out-of-the-box ScatterView. Once we learn the touch APIs in chapters 7 and 8, you'll be better prepared to implement that type of solution.

The ScatterView itself has only two methods beyond the standard ItemsControl. These methods on the ScatterView class are Activate(object item) and Deactivate(object item). You should pass the data-bound item or direct child that you want to activate or deactivate. You should not pass the ScatterViewItem unless you hard-coded an SVI as the item or child.

The ScatterViewItem also has two events that are fired when the SVI activation changes. These events are appropriately named ContainerActivated and ContainerDeactivated. We will see these two events in action later in this chapter.

Now we have covered how to create a data-bound ScatterView as well as all the features of the ScatterView and ScatterViewItem classes. With the knowledge of these classes, you can customize the behavior of your ScatterViewItems and access all of the information you need about when and how the user is interacting with the SVIs.

In addition to customizing the behavior of the ScatterView, we can also customize how the content is presented visually. The next section will cover a few common customization scenarios that are commonly requested in online forums and may come in handy in your applications.

6.3 Customizing the ScatterViewItem visual

You can actually use the out-of-the-box ScatterViewItem for a lot of different scenarios. As long as your content is rectangular or you are happy filling up a rectangular space, then you are good to go with the stock ScatterViewItem. There are a couple of scenarios where you might want to put some effort into customizing the ScatterViewItem visual.

The first scenario that we will cover is creating ScatterViewItems with round corners. This may be the case if you just want a more rounded feel to the interface, or if you actually have circular content that you want to use in the ScatterView.

The second scenario is if you want to change the activation effects and create a completely different look and feel. You may want to do this if you don't like the default shadow and sheen effects, or if you don't want the SVI to grow because the user needs to place them precisely.

The last scenario I will cover is how to smoothly animate ScatterViewItems in and out of existence. This is a very common request and is actually not that hard to achieve.

We will cover each of these scenarios, but let's begin by making your ScatterViewItems a little less pointy!

6.3.1 Creating ScatterViewItems with round corners

Let's dive right into the steps required to achieve rounded corners with a ScatterViewItem using the ExploringScatterView sample. The first thing we'll need to do is modify the StateView to have rounded corners. Open StateView.xaml and add the following attribute to the Border tag:

```
CornerRadius="20"
```

We will want to increase the StackPanel's Margin to 20 as well.

```
<StackPanel Margin="20">
```

Running the application now will produce a result like figure 6.9.

Figure 6.9 After adding rounded corners to the StateView, we see this as the result. Notice the ScatterViewItem and its shadow are still rectangular.

Even though the content is rounded, the SVI is still rectangular and its background fills the corners. We will also have to deal with the shadow, which is generated through a special control in the SVI control template.

CUSTOMIZING THE SVI STYLE

We can get rid of the SVI background (and border) by changing the style of the SVI. This is easiest to do in Expression Blend. If you have Blend, then open the project and then open SurfaceWindow1.xaml. Figure 6.10 shows what you need to click.

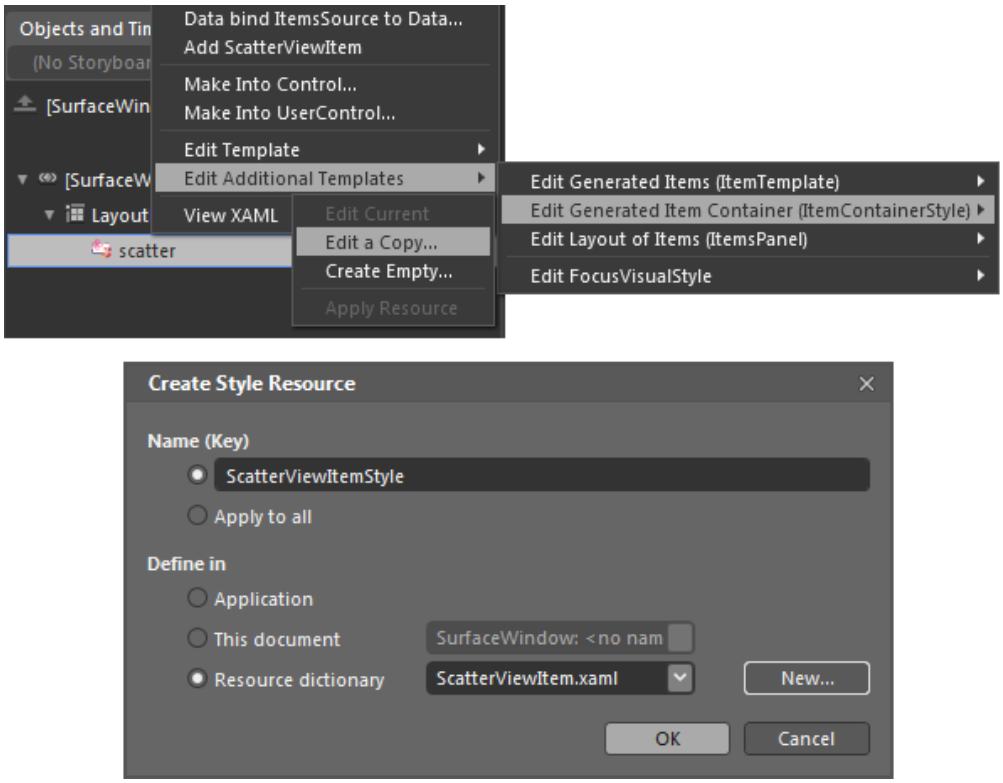


Figure 6.10 To customize the ScatterViewItem style, we need to create a copy. The top screenshot shows the right-click menu options to select on the scatter object in SurfaceWindow1.xaml. The bottom screenshot shows the options to select in the dialog that pops up. Note that you'll need to click "New…" to create the resource dictionary.

In the Objects panel, right click the scatter object and select "Edit Additional Templates", "Edit Generated Item Container (ItemContainerStyle)", then "Edit a Copy…" This will create a copy of the default SVI style. In the popup dialog, name the style "ScatterViewItemStyle", then click "New…" and create a new resource dictionary called ScatterViewItem.xaml. Click OK then OK again.

NOTE

It isn't necessarily required to place the custom style in its own resource dictionary. We could place it directly under the ScatterView, in Window.Resources, or elsewhere. I find that placing each style in its own resource dictionary doesn't bloat your main window file and makes it easier to read and find the resources you're looking for. It also makes it easier to reuse your style in other applications by simply copying and referencing the resource dictionary file.

If you do not have Blend, then you can still do this same technique manually. You'll need to add the ScatterViewItem.xaml resource dictionary to your project and reference it in the App.xaml merged dictionary, as shown in bold:

```
<Application.Resources>
    <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary
[CA]Source="/Microsoft.Surface.Presentation.Generic;v1.5.0.0;
[CA]31bf3856ad364e35;component/themes\styles.xaml"/>
        <ResourceDictionary Source="ScatterViewItem.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Next you'll need to add the copied style to the resource dictionary. I won't reproduce the style here, since it would take a few pages, but if you look in the provided ExploringScatterView sample code, I have

included the original style in the file ScatterViewItem.original.xaml, for reference. Finally, you would need to reference the new style by adding this attribute to the ScatterView tag in SurfaceWindow1.xaml:

```
ItemContainerStyle="{StaticResource ScatterViewItemStyle}"
```

Note that Blend will default to using DynamicResource, but in this case StaticResource will suffice. Now that we have a copy of the original ScatterViewItem style, we can modify it however we like. Let's start by getting rid of the background.

GETTING RID OF THE SVI BACKGROUND

Open ScatterViewItem.xaml and notice that it starts with two LinearGradientBrushes and three SolidColorBrushes. These are referenced from the main style and used for the background and border, plus alternate backgrounds and borders when the SVI is disabled. The main style starts with these lines:

```
<Style x:Key="ScatterViewItemStyle"
       TargetType="{x:Type Custom:ScatterViewItem}">
    <Setter Property="Background"
            Value="{StaticResource ScatterViewItem_GradientBrush}" />
    <Setter Property="BorderBrush"
            Value="{StaticResource ScatterViewItem_BorderBrush}" />
```

This is convenient because these are the first two things we need to change. Change the Value attribute of both Background and BorderBrush to:

```
Value="{x:Null}"
```

We want to use null here rather than a transparent SolidColorBrush. This is because transparent would still react to touches, while null does not. If we run the application now, we would see a result like figure 6.11.

Figure 6.11 The ScatterViewItem's gray gradient background is now gone, but we still have the shadow and the sheen to deal with. The left image shows the detail of a SVI corner at rest. Notice the shadow is visible. The middle image shows the rectangular sheen as the SVI as it activates. The right image shows the fully activated SVI and the obvious shadow.

Now the background is gone, but there are still some more rectangular visual elements. When the SVI is inactive, the shadow is hidden behind the foreground content. With content with round corners the shadow now sticks out. In addition, the visual sheen that animates as the SVI activates and deactivates is also rectangular.

ROUNDING THE SHADOW AND SHEEN

To make the shadow and the sheen have rounded corners, we will need to change a few parts of the control template. Find the SurfaceShadowChrome tag and replace it with this XAML:

```
<Microsoft_Surface_Presentation_Generic:SurfaceShadowChrome
                            x:Name="shadow"
                            Color="Transparent"
                            IsHitTestVisible="False"
                            ShadowVector="{TemplateBinding ShadowVector}">
    <Rectangle x:Name="ShadowRectangle"
               RadiusX="20"
               RadiusY="20"
               HorizontalAlignment="Stretch"
               VerticalAlignment="Stretch"
               Margin="4"
               Fill="#26000000" />
</Microsoft_Surface_Presentation_Generic:SurfaceShadowChrome>
```

This will render the shadow using this Border with rounded corners, but there are two animations which we need to update. A bit above the SurfaceShadowChrome you will find an ActivationTimeline Storyboard

and a DeactivationTimeline Storyboard. Each of these Storyboards has a ColorAnimationUsingKeyFrames entry that we need to update. Simply change the opening tag to:

```
<ColorAnimationUsingKeyFrames BeginTime="00:00:00"
                              Storyboard.TargetName="ShadowRectangle"
                              Storyboard.TargetProperty="Fill.Color">
```

This ensures that the activation and deactivation animations changes the color on the new rectangle, not the SurfaceShadowChrome, which now has Color="Transparent".

Finally, we need to update the Sheen. This is simple. A little below the SurfaceShadowChrome there will be another Rectangle with the name of "Sheen". Add a RadiusX and RadiusY property to it:

```
<Rectangle x:Name="Sheen"
           RadiusX="20"
           RadiusY="20"
           [...]>
```

At this point, if you run the application the ScatterViewItem shadows and sheen will be rounded and everything will look great, as in figure 6.12.

Figure 6.12 Corner detail of a ScatterViewItem with rounded content, shadows, and sheen. On the left is the inactive SVI with no rectangular shadow showing. On the right is the activated SVI with a rounded shadow. The sheen is also rounded, but not visible here.

This works, but there is still one small issue that bugs me. If we really wanted to reuse this resource dictionary in different scenarios, we would have to change the radius values within the resource dictionary per application. In an ideal world, we would be able to use the same resource dictionary and set the corner radius per ScatterView. Even better would be setting it dynamically per ScatterViewItem.

DYNAMICALLY SETTING THE CORNER RADIUS

Actually, it is pretty easy to dynamically set the corner radius through the magic of the DynamicResource. DynamicResource lets you set a reference to a named resource in XAML but does not look up the resource until it actually needs the value. In ScatterViewItem.xaml, change the two rectangles, ShadowRectangle and Sheen, so their RadiusX and RadiusY properties look like this:

```
RadiusX="{DynamicResource SVIShadowRadius}"
RadiusY="{DynamicResource SVIShadowRadius}"
```

This avoids hard-coding the values and makes it look up the resource at runtime. If it does not find the value, it will default to zero.

Now we just have to provide the resource. We could provide it with the ScatterView or Window resources, but actually the best place to provide it is in the StateView, where we define the other radius.

Open StateView.xaml.cs and at the end of the ConfigureSVI() method, add this line:

```
svi.Resources.Add("SVIShadowRadius", 20.0);
```

We already had a reference to the SVI associated with the StateView, so this ends up being a very elegant solution. Each individual SVI will look up the key and find it in its own resource dictionary.

If you needed to, you could vary the radius per item. In fact, I tested this initially by setting the radius to a random value. In a real application, you might have different types of data with possibly different radius values in the same ScatterView. This technique will allow you to reuse the same style for all of them and even set the radius based upon stored data, if you needed to.

NOTE

Since we are going to continue to change and evolve the sample project, I have provided a snapshot of the current status of the ExploringTouchControls project in the ExploringTouchControls.RoundCorners directory.

We have thoroughly covered creating rounded SVIs while preserving all of the activation effects. What do you do if you don't like the activation effects? That brings us to the next scenario.

6.3.2 Customizing ScatterViewItem activation effects

In the most basic sense, if you just want to get rid of the activation effects of growing, shadow, and visual sheen, then you could set `ShowsActivationEffects = false` on the ScatterViewItem. There will be no visual feedback of the activation, though, unless you provide it.

Since we have already explored the SVI style and control template, you could also customize the visuals in there and how the animations work. This would potentially be a reusable solution, but it is somewhat error-prone. The template uses triggers rather than the more modern VisualStateManager, so you might want to start from scratch.

Rather than going down that route, I want to explore another option that enables you to integrate the activation effects into the actual content. For example, suppose that you wanted the title text to subtly change style or the background color of your content to change. Those would be difficult to implement just by changing the SVI style.

In our case, we're going to change the style of our application a bit and create a more modern, minimalistic style. Figure 6.13 shows the end result of what we will do.

Figure 6.13 We will create these activation effects instead of the default SVI effects. The top-left image shows the inactive state. The top-right image shows the animation during activation. The bottom-left image shows the activated state. The state label animates out from under the flag and a border fades in.

Instead of growing and showing a shadow and sheen, we will make the inactive control state be just the flag. When the user touches the flag, the SVI activates and the StateView control animates the title out from under the flag and fades in a border. Let's quickly go through the modifications required.

CREATING AN EMPTY SVI STYLE

In SurfaceWindow1.xaml, change the background of the LayoutRoot Grid to White. In the SurfaceWindow.Resources, add this style:

```
<Style x:Key="EmptyScatterViewItemStyle"
       TargetType="{x:Type s:ScatterViewItem}">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type s:ScatterViewItem}">
                    <ContentPresenter />
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Then reference this new style by changing the ItemContainerStyle of the ScatterView to this:

```
ItemContainerStyle="{StaticResource EmptyScatterViewItemStyle}"
```

This effectively resets the entire style of the ScatterViewItems. They will still pan, rotate, and scale, as well as track the activation status, but they will not have any visual of their own. We could have just created a style that set ShowsActivationEffects to false, but we would still have the shadow under our control. This approach ensures we have a clean slate to start our custom activation visuals.

SETTING UP NEW ACTIVATION STATES

Now we need to add the activation visual ourselves in the StateView control. We will be using the VisualStateManager to control the visual states.

VisualStateManager

The VisualStateManager (VSM), now available for WPF in .NET 4, tracks the states and logic for transitioning between different visual states on a control. It was originally developed for Silverlight in the Silverlight Toolkit. It proved to be a popular way to develop controls, compared to triggers and setters, and it was added to the core Silverlight framework. Later the VSM was ported into the WPF Toolkit and now is available in WPF proper with .NET 4.

Think of the different visual states of a control, such as a button. All of the default settings in the visual of the button serve as the base state. Control authors can then define several visual states, such as pressed, or disabled, and specify the transition from the base state to the alternate states through StoryBoard animations. For example, the pressed state may require a transition for a specific property of a particular element in the control template.

Control authors can trigger the transition by setting the VSM state. The VisualStateManager will then handle activating the appropriate storyboards and animations. When during the transition, any element properties not explicitly set in the state's StoryBoard will transition back to the base state.

Open StateView.xaml and delete everything inside of the LayoutRoot Grid then add the contents of listing 6.5.

Listing 6.5 Updated StateView.xaml enables custom activation effects

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="ActivationGroup">
        <VisualStateGroup.Transitions>
            <VisualTransition GeneratedDuration="0:0:0.3">
                <VisualTransition.GeneratedEasingFunction>
                    <CircleEase EasingMode="EaseInOut" />
                </VisualTransition.GeneratedEasingFunction>
            </VisualTransition>
        </VisualStateGroup.Transitions>
        <VisualState x:Name="Inactive" />
        <VisualState x:Name="Active">
            <Storyboard>
                <ColorAnimationUsingKeyFrames Storyboard.TargetProperty=
                  [CA]"(Border.BorderBrush).(SolidColorBrush.Color)"
                      Storyboard.TargetName="border">
                  <EasingColorKeyFrame KeyTime="0"
```

```
                                    Value="Black" />
                </ColorAnimationUsingKeyFrames>
                <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty=
                  [CA]"(UIElement.RenderTransform).(TranslateTransform.Y)"
                      Storyboard.TargetName="textBlock">
                    <EasingDoubleKeyFrame KeyTime="0"
                                          Value="-50" />
                </DoubleAnimationUsingKeyFrames>
            </Storyboard>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

These VisualStateManager animations were generated using Blend, but the XAML is not too complex to understand. In short, when the VisualStateManager's state is set to Active, this XAML will animate the border's BorderBrush from its default color to Black and translates the textBlock by changing its TranslationTransform. When the state is set to Inactive, everything will animate to its default value. These transitions will take only three-tenths of a second and will use easing so the starts and stops will not be abrupt.

The rest of the LayoutRoot Grid content is shown in the rest of listing 6.5 below.

Listing 6.5 continued

```
<Border x:Name="border"
        BorderBrush="#00000000"
        BorderThickness="2"
        CornerRadius="10">
    <Viewbox x:Name="viewbox">
        <Grid Margin="10">
            <TextBlock x:Name="textBlock"
                       Text="{Binding StateName}"
                       HorizontalAlignment="Center"
                       FontSize="24"
                       VerticalAlignment="Top">
            <TextBlock.RenderTransform>
                    <TranslateTransform />
            </TextBlock.RenderTransform>
            </TextBlock>
            <Image x:Name="image"
                    Source="{Binding FlagURI}"
                    VerticalAlignment="Top" />
        </Grid>
    </Viewbox>
</Border>
```

This XAML is mostly the same as the original StateView XAML. The main differences are that the Border's BorderBrush is transparent and the TextBlock and Image are arranged inside of a Grid instead of a StackPanel. This will hide the text by default. The result is the default visual just shows the flag.

TRIGGERING THE VISUAL STATE CHANGES

We now need to trigger the state change when the container is activated. Open StateView.xaml.cs and find the ConfigureSVI() method. We need to update the SVIShadowRadius and add a few more lines to the end of this method. Update ConfigureSVI() to include listing 6.6.

Listing 6.6 Code to trigger VSM states when SVI is activated

```
void ConfigureSVI()
{
    […]
    svi.Resources.Add("SVIShadowRadius", 10.0);

    svi.ContainerActivated += (s, e) =>
    {
        VisualStateManager.GoToState(this, "Active", true);
    };

    svi.ContainerDeactivated += (s, e) =>
    {
```

```
            VisualStateManager.GoToState(this, "Inactive", true);
        };
    }
```

This code updates the SVIShadowRadius to 10.0, which matches the new XAML from listing 6.5. It also adds event handlers for the ScatterViewItem's ContainerActivated and ContainerDeactivated events. These event handlers simply tell the VisualStateManager to go to the appropriate state. The ContainerActivated and ContainerDeactivated events are called when the normal activation effects would be triggered, even if we have disabled those effects by using a minimal SVI control template.

That's it! Now if you run the application, you have custom activation effects and the overall look and feel will look like figure 6.14.

Figure 6.14 The modified ExploringScatterView sample with a white background, minimal visual elements, and custom ScatterViewItem activation effects.

The final result has a minimalistic visual style and only shows the title when you have activated the ScatterViewItem. This specific animation may be useful for a flash-card type of application where the user is memorizing the state flags. The general technique and approach might be useful for showing contextual controls only when the user is interacting with an element. This fits into the contextual concept of direct interaction.

NOTE

A snapshot of the sample project in its current state is available in the ExploringTouchControls.ActivationEffects directory.

So far our sample application has shown all of the ScatterViewItems from the start of the application and the positions and orientations are randomly determined by the ScatterView. If you had backend code that was adding or removing data, or if the user requests additional data, right now the SVI will just pop

into existence at a random location. This is often an undesired effect. In the next section, we will take a look at an easy way to smoothly animate ScatterViewItems in and out of existence.

6.3.3 Fluid ScatterViewItem transitions

When a ScatterViewItem is added to a ScatterView, it defaults to a random position and orientation. We have already seen where we can initialize the position and orientation in the ConfigureSVI() method. Suppose that we want to create a fluid visual experience for the user when these SVIs appear or disappear. What is the best way to do that?

Actually, the easiest way to do it is to animate the Center and Orientation properties of the ScatterViewItem. These are both DependencyProperties, which means they can take part in data binding as well as Storyboards. It might be possible to create some Storyboards in XAML that modify these properties but it would involve some level of indirection and may not be flexible enough for many situations.

I come across this situation is almost every application, so I ended up writing a utility class that helps me dynamically create animations on any type of property. I added another method that lets me animate ScatterViewItems to a new position and orientation with one method call. This method is called SurfaceAnimateUtility.ThrowSVI() and it is available in the Blake.NUI library at http://blakenui.codeplex.com. Table 6.9 shows the parameters of the ThrowSVI method.

Table 6.9 ThrowSVI takes these parameters

Parameter	Description
ScatterViewItem svi	The ScatterViewItem to animate
Point targetPoint	The SVI will be moved to this position
double targetOrientation	The SVI will be rotated to this orientation
double fromTime	The animation will start this many seconds from now
double toTime	The animation will complete this many seconds from now
IEasingFunction ease = null	Optional, specify an easing function to use for the animation.

ThrowSVI returns an AnimationClock that lets you monitor the progress of the animation. You can subscribe to AnimationClock's Completed event to be notified when the animation is complete. Let's see how we can use this method.

ANIMATING THE SVI POSITIONS

We are going to add just a couple lines of code that will make the ScatterViewItems start off-screen and then animate on-screen into a grid. In the DataControls project, add a reference to Blake.NUI.WPF.dll and Blake.NUI.WPF.SurfaceToolkit.dll. Next, open StateView.xaml.cs and add these two using statements at the top:

```
using System.Windows.Media.Animation;
using Blake.NUI.WPF.SurfaceToolkit.Utility;
```

Now find the ConfigureSVI() and update and add the code shown in listing 6.8 to the end of the method.

Listing 6.8 Start SVIs off-screen then animate on-screen

```
void ConfigureSVI()
{
    [...]
    svi.Height = 100;                                          #A
    [...]

    Point targetPoint = new Point(100, 100);

    svi.Orientation = -90;
    svi.Center = new Point(targetPoint.X, -300);               #B
```

```
        BackEase backease = new BackEase();
        backease.EasingMode = EasingMode.EaseOut;
        backease.Amplitude = 0.3;

        double startTime = 1.0;

        SurfaceAnimateUtility.ThrowSVI(svi,                    #C
                                       targetPoint,            #C
                                       0.0,                    #C
                                       startTime,              #C
                                       startTime + 1,          #C
                                       backease);              #C
    }
```

#A Smaller initial size
#B Start off-screen
#C Animate to targetPoint

First, we want to make the default size a bit smaller than before, so we decrease the size to This code will set the ScatterViewItem's initial position to be above the top of the window. It then sets up an easing function for the animation and then animates the SVI to the target position and orientation.

Notice that because the start time is 1.0, the animation has a slight delay. This gives the application a chance to finish loading and displaying before animating anything.

ARRANGING SCATTERVIEWITEMS INTO A GRID

As is, this just animates all the SVIs to the same position, which is pretty boring. Let's add a few more lines that will give us a nice grid.

> **NOTE**
>
> This next section of sample code is a bit of a hack. In a real application, you would want to have a ViewModel or a service determine the grid positions in a robust way. For the purposes of getting through this sample, I'm just going to provide a quick way to get it done. This works well for a screen resolution of 1280x800 or larger, but will be cut off on smaller screens.

At the beginning of the StateView class, add these lines:

```
static Stack<Point> availablePositions = new Stack<Point>();
static StateView()
{
    for (int i = 1275; i >= 75; i -= 150)
    {
        for (int j = 675; j >= 75; j -= 100)
        {
            availablePositions.Push(new Point(i, j));
        }
    }
}
```

This creates a collection of grid positions. I realize this is rather inelegant, but we just need to produce some sample data to test. A real application would get the positions from a data store or else generate the grid positions in a more robust way.

We get these grid positions by adding these two bold lines after we declare targetPoint:

```
Point targetPoint = new Point(100, 100);
if (availablePositions.Count > 0)
    targetPoint = availablePositions.Pop();
```

Now change the startTime declaration to this line:

```
double startTime = 1 + (700 - targetPoint.Y) / 700 + targetPoint.X/2000;
```

This calculates an offset startTime based upon the target position of the SVI. The end result is a staggered waterfall of flags, shown in figure 6.15.

Figure 6.15 Flags now fall onto the screen into a grid layout. The animation is staggered from bottom-left to top-right. The BackEase function gives the animation a little overshoot.

This same technique can be used to animate ScatterViewItems from a particular point on the screen, such as a data source like a map pushpin or an object that represents a mobile device, to either a freeform or an organized layout. You can also animate objects out of existence by sending ScatterViewItems off-screen or towards a data store while turning transparent.

We have now covered the ScatterView in detail. ScatterView is useful for freeform interaction as well as searching and sorting a limited amount of content. The Library controls can be used with the Scatterview to more organization than just the ScatterView alone. In the next chapter, we will extend this sample application to use the Library controls and also see how we can enhance the interface with the ElementMenu and TagVisualizer controls.

6.4 Summary

In this chapter we learned about the ScatterView control and saw how to use it in a real-world data binding scenario. We also saw how to customize the look and feel of ScatterViewItems and integrate the activation behavior with the behavior of your own controls.

With this knowledge, you can create simple multi-touch applications. The ScatterView is most useful for loosely related content. For other types of content relationships, we need other containers. In the next chapter, we will cover the library controls, which are great starting points for containers and are most useful for content that falls into categorizes and sorting activities. We will also learn several other controls designed specifically for touch and Surface: the Library controls, ElementMenu, and TagVisualizer.

7 Learning new Surface SDK controls

In the last chapter, we focused on the ScatterView control. That control is just one of several new controls in Surface SDK that are designed for touch. In this chapter, we will continue the ExploringTouchControls example and add the LibraryStack, LibraryBar, and LibraryContainer to it. These controls provide some more variety to the ways that we can organize content into containers.

We will also learn about the ElementMenu and the TagVisualizer controls. *More details on these controls excluded from the MEAP due to NDA restrictions.*

To get started, let's learn about the Library controls.

7.1 Library controls

Surface SDK provides three more controls designed for multi-touch: LibraryStack, LibraryBar, and LibraryContainer. Within the OCGM model, these control are considered containers, as the names imply. They serve as containers for other content and arrange the content in a specific way. These controls also have drag-and-drop built in to make it extremely easy to move content from one library control to another.

In this section, we will talk about each of these controls and see how to use them in the ExploringTouchControls sample application. Let's start with the LibraryStack.

7.1.1 LibraryStack

The LibraryStack arranges content in a stack with a slightly disorganized but non-chaotic look. The content is all located at the same location with about the same size, but with slightly different rotations.

The LibraryStack can be used as a location for categorizing content or as a way to select one item out of a small collection of items. For example, you could use the LibraryStack to let used add and remove products from a shopping cart, or shuffle through a set of photos to select an individual picture. Figure 7.1 shows a LibraryStack populated with state flags.

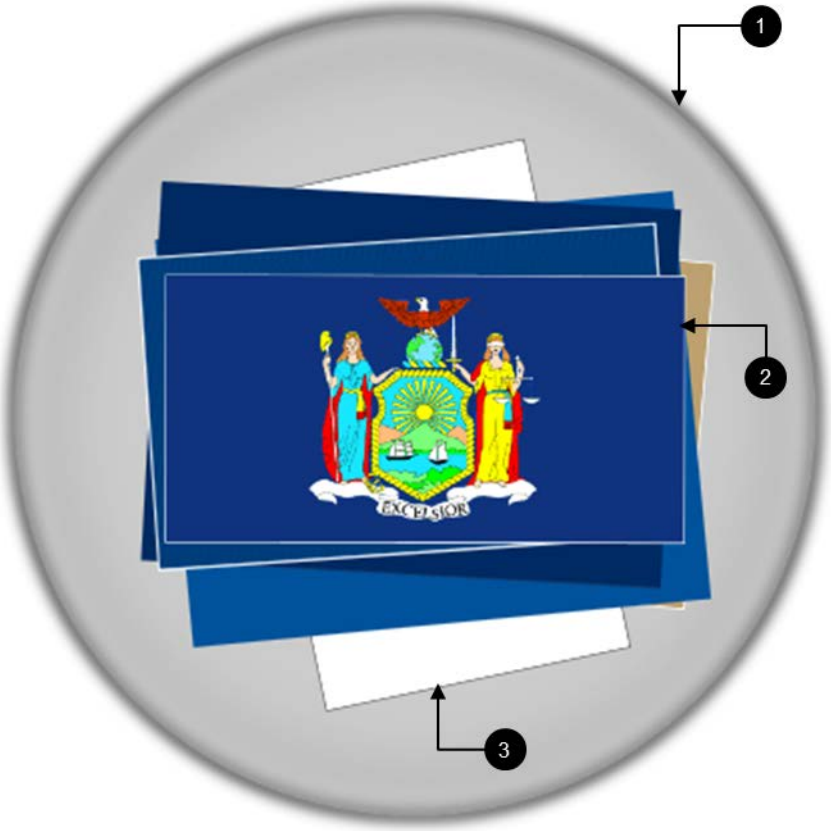


Figure 7.1 The LibraryStack organizes content in a stack. The outer circle and background #1 serves as a handle for moving the entire stack as well as for dropping items into the stack. When you tap the front item #2, it will animate to the back of the stack. The item at the back of the stack #3 is rotated more than the others so you can tap it to bring it back to the front.

As you can see, items have a randomized orientation, but the randomization is limited to a small range. Two positions have special orientation values. The front item has no rotation and the back item has almost ninety degrees rotation.

The LibraryStack can be used for You can cycle through the stack by tapping the front #2 or back #3 item.

USING THE LIBRARYSTACK

You can add the LibraryStack and other library controls anywhere in your window, though they are designed to be used inside of a ScatterView. In ExploringTouchControls, we already have a ScatterView, but the ItemsSource is set to bind it to the States collection. When the ItemsSource is set on an ItemsControl, such as ScatterView, we cannot manually add items.

We will be coming back to this when we talk about drag-and-drop in chapter 10, but for now let's just create a new ScatterView that will hold our library controls. Open SurfaceWindow1.xaml, comment out the existing ScatterView and then add listing 7.1.

Listing 7.1 Adding a LibraryStack to a ScatterView

```
<s:ScatterView Name="libraryScatter">
    <s:ScatterViewItem Style="{StaticResource
        [CA]LibraryControlInScatterViewItemContentStyle}"
                       Orientation="0.0"
                       Center="850,400">
            <s:LibraryStack Width="300"
```

```
                        Height="300">
                    <Rectangle Width="200" Height="100" Fill="Red" />
                    <Rectangle Width="200" Height="100" Fill="Blue" />
                    <Rectangle Width="200" Height="100" Fill="Green" />
                    <Rectangle Width="200" Height="100" Fill="Yellow" />
                    <Rectangle Width="200" Height="100" Fill="Orange" />
                </s:LibraryStack>
            </s:ScatterViewItem>
        </s:ScatterView>
```

Just like the ScatterView, we can manually add children to the LibraryStack and other library controls. In this case, we've added a few colored rectangles. We could have placed the LibraryStack as a direct child of the ScatterView, but then we would have no control over its initial position and style. In this case, we provide the ScatterViewItem for the LibraryStack so that we can set the initial position and orientation of the SVI.

Library Control in ScatterViewItem styles

You may also notice the SVI style is set to LibraryControlInScatterViewItemContentStyle. This style creates an empty SVI template and set up binding between the library control width and height and the SVI width and height. Without it the LibraryStack will be contained within a square SVI with the wrong size. This style comes from the LibraryControlResourceDictionary.xaml resource dictionary, which is distributed with the Surface SDK sample code in the ControlsBox sample.

If you have not already unzipped the Surface Code Samples.zip, you can find it in the start menu under Microsoft Surface/samples. Unzip it to somewhere in your documents, rather than in place. The LibraryControlResourceDictionary.xaml file is located within these samples in the folder: Surface Code Samples\SDKSamples\WPF\Shared\.

I have provided the file in the complete sample code, but if you are adding it manually to this project or another project, you will also need to reference the resource dictionary in App.xaml:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            [...]
            <ResourceDictionary
                Source="LibraryControlResourceDictionary.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

The referenced resource dictionary defines the LibraryControlInScatterViewItemContentStyle style, which should be used for ScatterViewItems around LibraryStacks and LibraryBars, as well as the LibraryContainerInScatterViewItemStyle style, which should be used for ScatterViewItems around LibraryContainers.

Run the program and try out the behavior when you tap the front item or the back item. Also notice that you can drag the rectangles out of the stack and rotate and scale them. This is the integrated drag-and-drop behavior. For now there is nowhere to drop it, so when you release the rectangle it will animate back to the top of the stack.

There are also two very subtle behaviors. The first occurs if you touch and hold an item or if you drag an item away from the stack then back over it. The edge of the LibraryStack visual will animate as shown in figure 7.2.

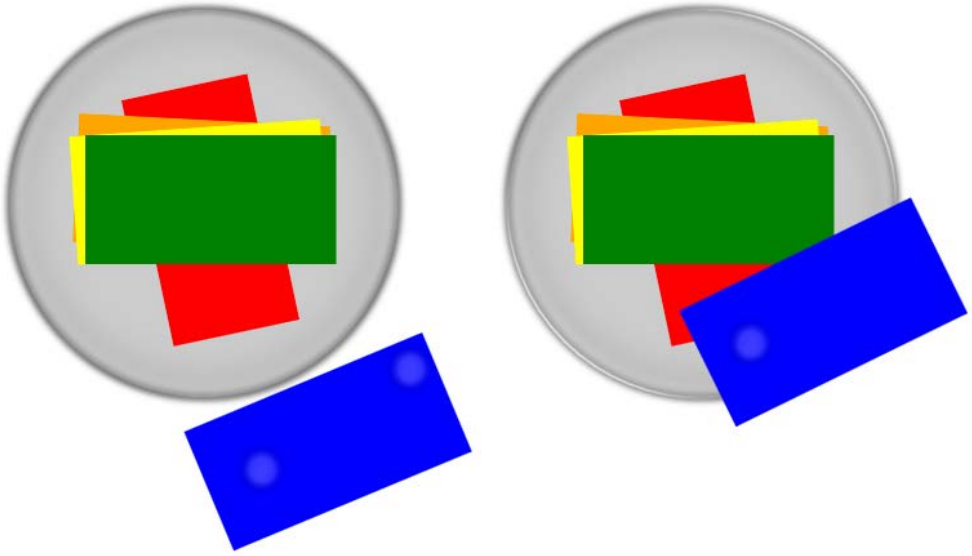


Figure 7.2 A simple LibraryStack with hard-coded rectangles. The left image shows how you can drag the rectangles out and rotate them similar to a ScatterViewItem. The right image shows the change in the LibraryStack visual when an item can be dropped into it.

This animation indicates to the user that the object being manipulated over the stack can be dropped into it. The other library controls also have a similar animation.

The second subtle behavior occurs if you drag and drop an item only a short distance from its initial position without ever leaving the bounds of the stack. In this case, instead of the rectangle dropping to the top of the stack, it will drop to the back of the stack the same as if you just tapped it. You can tell if the rectangle leaves the stack because the animation in figure 7.2 will go away.

This drop-to-back behavior allows users a little bit of flexibility if they are quickly tapping through the stack but accidentally move an item slightly. The item will animate to the back of the stack as the user likely expects it to. It is the kind of thing that users might never consciously notice but would cause frustration if it wasn't there. This is the result of good design and focusing on the user experience.

DATA BINDING THE LIBRARYSTACK

Now that we have seen how the LibraryStack behaves, let's bind it to some data. In SurfaceWindowViewModel, we have a States collection that provides values for the ScatterView. Since we have created another container, we will need another collection. We may use the States collection again in the future. Add the following code inside the Properties region of SurfaceWindowViewModel:

```
ObservableCollection<StateModel> _stackStates =
    new ObservableCollection<StateModel>();
public ObservableCollection<StateModel> StackStates
{
    get { return _stackStates; }
}
```

We will bind our LibraryStack to this collection, so we need to make sure it has some data. In the InitData() method change this line:

```
models.ForEach(States.Add);
```

to this:

```
models.ForEach(StackStates.Add);
```

Now our new collection will be populated with data. We just need the LibraryStack to bind to it. In SurfaceWindow1.xaml, add the following data template to Window.Resources:

```
<DataTemplate x:Key="StateTemplateStack">
    <Viewbox>
        <Image Source="{Binding FlagURI}" />
    </Viewbox>
</DataTemplate>
```

This is a minimal template that will just display the flag image.

NOTE

Normally I would simply reuse the StateTemplate data template rather than creating a new one, but in testing this application with the Surface Toolkit beta, there was an odd behavior. When I used the StateView inside the data template for a LibraryStack, the shuffle and drop animations had incorrect size transitions. For now, I'm providing an alternative that does not have this problem. If I discover a better solution or the final version of Surface Toolkit does not have this problem, I will update this section before the final printing.

Next, update the LibraryStack that we added from listing 7.1 to this:

```
<s:LibraryStack Width="300"
                Height="300"
                ItemsSource="{Binding StackStates}"
                ItemTemplate="{StaticResource StateTemplateStack}" />
```

Here we are binding the items to the new StackStates collection and telling it to display those items using the new data template we just added. If you run the application now, you will see the LibraryStack bound to the flag images, as in figure 7.3.

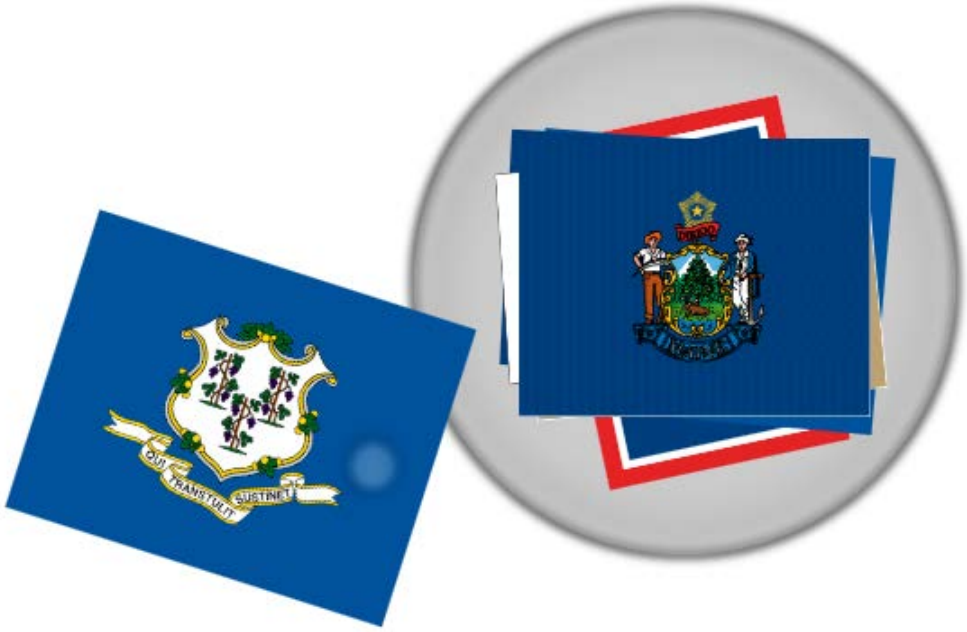


Figure 7.3 A LibraryStack bound to a collection of StateModels. I have dragged and rotated the top item off of the stack.

Now that we have our content in the stack, let's discuss the couple of other features it has.

LIBRARYSTACK INTERFACE

I mentioned that you can use the LibraryStack to allow users to shuffle through and select a specific item. The item on the top of the stack is the selected item. It would be helpful to know how to tell which item is selected.

LibraryStack derives from Selector, which is also the base class for ListBox. This means that determining the selected item with the LibraryStack is pretty much the same as ListBox. The most relevant of these would be the SelectedIndex and SelectedItem properties and the SelectionChanged event.

If you need to take action when the user selects an item, you can subscribe to the SelectionChanged event and check the SelectionChangedEventArgs or the SelectedIndex and SelectedItem. The

SelectedItem will return the direct child or the data bound object, such as the StateModel, of the item on top of the stack.

Aside from checking the selected item, there are only two other properties on the LibraryStack, listed in table 7.1.

Table 7.1 LibraryStack properties

Property	Description
DisplayItemCount	The number of items actually displayed at a single time. This value is read-only and is set to six. LibraryStack can contain any number of items but visually hides all but the top six for performance reasons.
DragCursorStyle	The style to use for the drag cursor. The drag cursor is displayed whenever an item is touched, even briefly during a tap. This allows you to change the visual style of a dragged item to look different or even show more or less information than when in the stack.

We will see how to use the DragCursorStyle as we learn about the LibraryBar.

We have seen how and when to use the LibraryStack, how to setup data binding, and how to check which item is selected. In the next section, we will jump right into the LibraryBar. The LibraryBar is similar to the LibraryStack, but useful in different scenarios.

7.1.2 LibraryBar

The LibraryBar lays out content in a horizontal scrolling grid. It also has the ability to group items according to categories you specify. LibraryBar is useful for browsing larger number of items, whereas LibraryStack is best suited for a smaller number of items.

Since we've already learned about the basics with LibraryStack, let's jump right into adding the data-bound LibraryBar.

DATA BINDING THE LIBRARYBAR

First we will need to add another collection to provide values to the LibraryBar. Open SurfaceWindowViewModel.cs and add this code to the Properties region:

```
ObservableCollection<StateModel> _barStates = new ObservableCollection<StateModel>();
public ObservableCollection<StateModel> BarStates
{
    get
    {
        return _barStates;
    }
}
```

Now change the InitData method to populate the new BarStates collection instead:

```
models.ForEach(BarStates.Add);
```

Open SurfaceWindow1.xaml. We are going to need two more resources so add listing 7.2 to Window.Resources. The StateTemplateBar data template will be used as the LibraryBar's ItemTemplate and will show the flag image below the name of the state. The StateDragStyle will be used when you drag an item from the LibraryBar and just shows the flag image.

Listing 7.2 New resources to support the LibraryBar

```
<DataTemplate x:Key="StateTemplateBar">
    <Viewbox>
        <Grid Width="160"
              Height="140"
              Margin="0,0,5,0">
            <Grid.RowDefinitions>
                <RowDefinition Height="32" />
                <RowDefinition Height="108" />
            </Grid.RowDefinitions>
            <TextBlock Text="{Binding StateName}"          #A
                       HorizontalAlignment="Left"
                       VerticalAlignment="Bottom"
```

```
                        Foreground="Black"
                        Margin="0,0,0,2"
                        FontSize="16" />
                <Image Source="{Binding FlagURI}"             #B
                       Grid.Row="1"
                       Stretch="Uniform"
                       VerticalAlignment="Top"
                       HorizontalAlignment="Left" />
            </Grid>
        </Viewbox>
    </DataTemplate>

    <Style x:Key="StateDragStyle"
            TargetType="ContentControl">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate>
                    <Viewbox>
                        <Image Source="{Binding FlagURI}"     #C
                               Stretch="Uniform" />
                    </Viewbox>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

#A Display state name
#B Display flag
#C Only show flag

Finally we need to add the LibraryBar itself. Add Listing 7.3 inside of the libraryScatter ScatterView.

Listing 7.3 The data-bound LibraryBar inside a ScatterViewItem

```
    <s:ScatterViewItem Style="{StaticResource
        [CA]LibraryControlInScatterViewItemContentStyle}"
                       Orientation="0.0"
                       Center="350,500">
        <s:LibraryBar Width="600"
                      Height="300"
                      MaxHeight="400"
                      MaxWidth="800"
                      Rows="1"
                      Foreground="Black"
                      ItemsSource="{Binding BarStates}"
                      DragCursorStyle="{StaticResource StateDragStyle}"
                      ItemTemplate="{StaticResource StateTemplateBar}" />
    </s:ScatterViewItem>
```

This XAML creates a SVI using the custom library control style we discussed, and inside the SVI is a LibraryBar. We configured the LibraryBar to use the new BarStates collection as well as the two resources we just added. If you run the application now you'll see something like figure 7.4.

Figure 7.4 A LibraryBar showing state flags. The ItemTemplate shows the name of the state as well as the flag image. I have dragged an item upwards out of the control and rotated and scaled it. The dragged item uses the DragCursorStyle, which does not include the state name.

You can scroll the LibraryBar by touching in the item area and moving left and right. You can also move, rotate, and scale the container itself by touching the gray areas on the top and bottom. If you touch an item and move up or down, you will drag the item out of the container.

Notice that the items in the LibraryBar show both the flag image and the state name. This is the result of the StateTemplateBar data template. If you drag an item out you will see the dragged item does not have the state name because the DragCursorStyle only includes the flag image in the control template.

As you can see in figure 7.4, the source item is still a member of the LibraryBar but is shown to be disabled by decreasing the opacity. Disabled items cannot be dragged again. The default drag-and-drop behavior for this control is to make a copy of the item and disable the original, but it can be customized once we learn more about the drag-and-drop framework. If you drop a copy back into the LibraryBar, it will enable again. You can test this by dropping an item in the LibraryStack then dragging it back to the LibraryBar.

You can check whether an item is disabled or enabled within a LibraryBar by calling the GetIsItemDataEnabled(object data) method. You can change this status using the SetIsItemDataEnabled(object data, bool isEnabled) method.

The LibraryBar is now ready to use, but we can customize it a little more. First, we can set Rows="2" to see two rows of items. You could set it to a higher number as well, but the items typically become so small they are unrecognizable. You could compensate for this by increasing the height of the LibraryBar, but the control's visuals were designed for a very wide aspect ratio and don't look good if the control is more square.

We can also group items into categories. This sounds difficult but it turns out to be rather easy to do with this control.

GROUPING LIBRARYBAR ITEMS

Depending upon the type of content displayed, you may want to group items into different categories. This is only possible when using data binding. The item grouping logic requires the use of a CollectionViewSource to configure what property contains the group descriptions.

Our data model class StateModel includes a Region property that would be a convenient grouping. Open SurfaceWindowViewModel.cs and add the following code at the end of the InitData method:

```
ICollectionView defaultViewBar =
    CollectionViewSource.GetDefaultView(BarStates);

defaultViewBar.GroupDescriptions.Add(
    new PropertyGroupDescription("Region"));
```

This code finds the default collection view for BarStates then adds a PropertyGroupDescription instance that identifies the Region property for grouping. As the LibraryBar is arranging items, it will group them according to the specified group description.

NOTE

Do not add more than one group description for collections bound to a LibraryBar. The results may be unpredictable as the control is only designed for zero or one group descriptions.

Figure 7.5 shows how the LibraryBar looks with group descriptions.

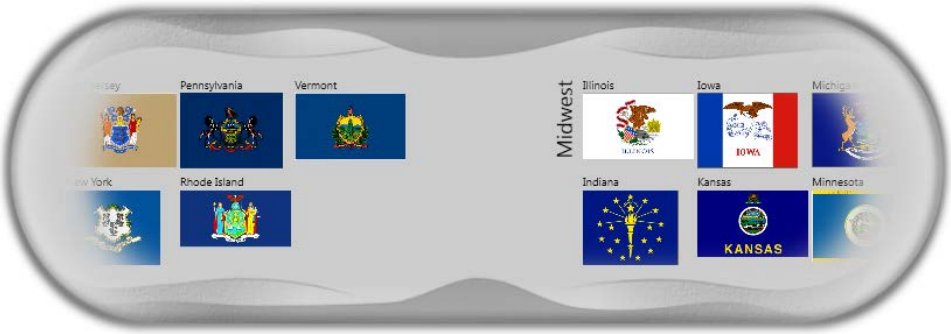


Figure 7.5 The LibraryBar with states grouped by region. This screenshot also shows the effect of setting Rows="2".

As you scroll the LibraryBar from one group of items to another, you can access the name of the currently displayed group by checking the LibraryBar's SelectedGroup property. You can also be notified when the group selection changes by subscribing to the SelectedGroupChanged event.

We have covered both the LibraryBar and the LibraryStack. They can both be used to sort, organize, and browse content. The LibraryStack is ideal for smaller groups of content and the LibraryBar works better with larger groups of content.

Since users can drag items in and out of these containers, the number of items can change and the original choice of LibraryStack or LibraryBar may not be ideal anymore. In those cases, it would be convenient to be able to dynamically switch between these two containers.

Fortunately, the Surface team has provided one more control that does exactly this: the LibraryContainer. We only have to discuss a few things about LibraryContainer because for the most part, it includes features you already know about from LibraryStack and LibraryBar. This is the last thing we will cover in this chapter.

7.1.3 Library Container

The LibraryContainer is basically a hybrid of LibraryStack and LibraryBar. It can change viewing modes between stack and bar mode. It transitions visually between these modes. Besides this transition, there are three things that the LibraryContainer has that the other two controls don't.

First, the LibraryContainer has extra visual handles on the sides. Second, the LibraryContainer has a category selector control on the top, provided item grouping is setup. Third, there is a small button on the bottom to enable switching between bar and stack modes. The LibraryContainers shown in figure 7.6 have these new visuals.

Figure 7.6 On the top is the LibraryContainer in stack mode and below is the same LibraryContainer after transitioning to bar mode. Notice the category selector control, additional visual handles, and the mode change button. I activated the category selector control on the bottom LibraryContainer.

Because the LibraryContainer can switch between both stack and bar modes, it is the most flexible of all the library controls and can be used for any amount of data. Let's see how to add LibraryContainer to our project, and then discuss the couple extra features of the LibraryContainer.

DATA BINDING THE LIBRARYCONTAINER

The first thing we need to do is create another collection of StateModels that will be the ItemsSource for the LibraryContainer. In SurfaceWindowViewModel.cs, add these lines in the Properties region:

```
ObservableCollection<StateModel> _containerStates = new
ObservableCollection<StateModel>();
public ObservableCollection<StateModel> ContainerStates
{
    get
    {
        return _containerStates;
    }
}
```

In InitData(), update the collection population line to:

```
models.ForEach(ContainerStates.Add);
```

Now add these lines in the same method:

```
ICollectionView defaultViewContainer =
    CollectionViewSource.GetDefaultView(ContainerStates);
```

```
defaultViewContainer.GroupDescriptions.Add(
    new PropertyGroupDescription("Region"));
```

This makes sure that the new collection will be ready to group items. Now you can add the actual LibraryContainer to the project. Open SurfaceWindow1.xaml and add the XAML in listing 7.4 to the end of the ScatterView.

Listing 7.4 Adding the LibraryContainer

```
<s:ScatterViewItem Style="{StaticResource
                    [CA]LibraryContainerInScatterViewItemStyle}"
                   Orientation="0.0"
                   Center="500,200"
                   Width="800"
                   Height="300"
    <s:LibraryContainer ViewingMode="Bar"
                        ItemsSource="{Binding ContainerStates}">
        <s:LibraryContainer.BarView>
            <s:BarView Rows="2"
                       DragCursorStyle="{StaticResource StateDragStyle}"
                       ItemTemplate="{StaticResource StateTemplateBar}" />
        </s:LibraryContainer.BarView>
        <s:LibraryContainer.StackView>
            <s:StackView DragCursorStyle="{StaticResource StateDragStyle}"
                         ItemTemplate="{StaticResource
                             [CA]StateTemplateStack}" />
        </s:LibraryContainer.StackView>
    </s:LibraryContainer>
</s:ScatterViewItem>
```

This XAML creates a ScatterViewItem that uses the LibraryContainerInScatterViewItemStyle. Note that this uses a different style than the one used for the LibraryBar and LibraryStack ScatterViewItems.

Notice that in the LibraryContainer tag, we only configured two attributes: the default viewing mode and the ItemsSource. We performed most of the rest of the configuration on the BarView and StackView properties using instances of a BarView and StackView. Here we can set the number of rows for the bar as well as the DragCursorStyle and ItemTemplate. We use a different ItemTemplate for the BarView and StackView, so when the user switches viewing modes by pressing the button on the bottom the individual items will change to the new template. Figure 7.6 showed the difference between the templates in each mode.

You can run the application now and test the behavior of the LibraryContainer. You can also drag items to and from the LibraryContainer in both stack mode and bar mode.

Notice that you can also switch between data groups in both stack and bar mode using the category selector control on the top of the LibraryContainer. When you shuffle through the stack, scroll the bar, or change the category selector, it changes the selected group. You can get access to the selected group and item information, but it is slightly different than the LibraryBar and LibraryStack equivalent properties.

LIBRARYCONTAINER INTERFACE

The LibraryContainer rolls a LibraryBar and LibraryStack into one and adds a few extra features. It exposes the individual features of LibraryBar and LibraryStack through slightly different names. Table 7.2 shows the features of LibraryContainer.

Table 7.2 LibraryContainer exposes the following events, methods, and properties

Name	Type	Description
SelectedGroupChanged	event	Fired when the user changes the category selector or pans the bar to a new group
StackSelectionChanged	event	Fired in stack mode when the top-most item changes
ViewingModeChanged	event	Fired when the LibraryContainer changes from stack to bar mode or bar to stack mode
GetIsItemDataEnabled	method	Gets whether a data item or child is enabled

SetIsItemDataEnabled	method	Sets whether a data item or child is enabled
BarView	property	Gets or sets the BarView so you can configure the bar mode options
CanSwitchViewingMode	property	Gets or sets whether the LibraryContainer can switch modes. When false, the mode switcher is hidden.
IsSelectorExpanded	property	Gets or sets whether the selector is expanded
IsTransitioning	property	Gets whether the LibraryContainer is animating between bar and stack mode.
SelectedGroup	property	Gets or sets the currently selected group
SelectedStackIndex	property	Gets or sets the index of the selected item in stack mode
SelectedStackItem	property	Gets or sets the currently selected item in stack mode
StackView	property	Gets or sets the StackView so you can configure the stack mode options
ViewingMode	property	Gets or sets the current viewing mode to either Bar or Stack.

As you can see, the LibraryContainer gives you the grouping of the LibraryBar and individual item selection of the LibraryStack.

The library controls are good for very quickly creating containers in your applications. They have complete visuals and include all of the necessary logic built-in to drag-and-drop between containers. When we discuss the drag-and-drop framework in chapter 10, we will see how to enable drag-and-drop between the containers and other controls such as SurfaceListBox or ScatterView.

7.2 ElementMenu

This section has been hidden from the MEAP due to NDA restrictions. It may be included in a future MEAP when the NDA is lifted. The final release of this book will include all content.

7.3 TagVisualizer

This section has been hidden from the MEAP due to NDA restrictions. It may be included in a future MEAP when the NDA is lifted. The final release of this book will include all content.

7.4 Summary

For more advanced applications, though, you'll need more than just ScatterView and the library containers. In the next few chapters, we will learn how to use the WPF Touch API so that you can create your own controls and containers. Once we have learned how to use raw touch and manipulations, we'll circle back to finish up the Surface SDK with the drag-and-drop framework and touch visualizations in chapter 10.

8

Accessing raw touch information

In the last few chapters, we talked about the Surface SDK controls and how those controls take advantage of touch. In many applications, you will need to go beyond the basic controls and create your own with custom behaviors that make sense for the particular application you are building. In order to create your own controls, you will need to know about the raw touch API and the manipulations API.

RAW TOUCH VERSUS MANIPULATIONS

There are two ways to access multi-touch data in WPF: the raw touch API and the manipulation API. In WPF, they aren't really separate APIs because both raw touch and manipulations are incorporated into the core presentation framework. I refer to them in this way though because they represent two entirely different ways to access and interpret touch information. The raw touch API gives you access to the low-level touch events and touch points, while the manipulation API interprets the low-level events, performs some math, and gives you access to high-level events that tell you what all the touches are doing in aggregate.

The most engaging natural user interfaces go beyond touch-enabled buttons, checkboxes, and radio buttons. Those controls have a place, but the most compelling NUIs have behavior and interface styles customized for specific content and interactions. You will need to know both the raw touch and manipulations APIs to create these customizations.

In chapter 9 we will look at the manipulations API, but to fully understand manipulations you need to first know about raw touch. This chapter covers the raw touch API in detail. We will see when you should use it, what methods and properties are in this API, and code samples of how to use raw touch in different scenarios. Let's start by talking about what raw touch is.

8.1 Overview of the raw touch API

In this section I'm going to give you a high-level view of what the raw touch API does, when and why you would want to use it, and what the components are. This will give you a good idea of what we're talking about when we dive into the specifics of the API and sample code throughout the rest of this chapter.

8.1.1 What does the raw touch API do?

The raw touch API gives you raw access to touch data and allows you to use that information however you see fit. The raw touch data allows your application to be aware of every individual contact point as well as any extra information that is made available by the hardware. Table 8.1 lists the information available through the WPF raw touch API.

Table 8.1 The raw touch API gives you access to raw data about individual touch events. This table shows touch-specific data that you have access to in any Windows 7 touch device.

Data	Description
Timestamp	When the touch event occurs
Action	Whether the current event is a touch down, move, or up event
Position	The position of the touch
Size	The area of the touch in contact with the touchscreen, such as the pad of the finger
Bounds	The bounds of the contact area calculated using the position and size

This information is provided by the touchscreen hardware and you can use it in a variety of ways. In section 8.2 we will discuss specifics about how to get to this data and in section 8.3 we will see some sample code of what we can do with it.

Touches have size

You may wonder why the size of the contact point is reported. Unlike a mouse cursor, which is a single pixel, a finger contact covers several pixels on the screen. Touch hardware typically detects the full area in contact and then filters the data to find the position. The size and bounds is useful in some scenarios because you can use the size to differentiate between different types of touches, such as a small round touch from the fingertip from a very long touch from the side of the hand. The sizes can also be used in touch visualizations and finger painting. To some extent, you can use the size of the touch to determine the pressure or the pose of the hand, as shown in Figure 8.1.

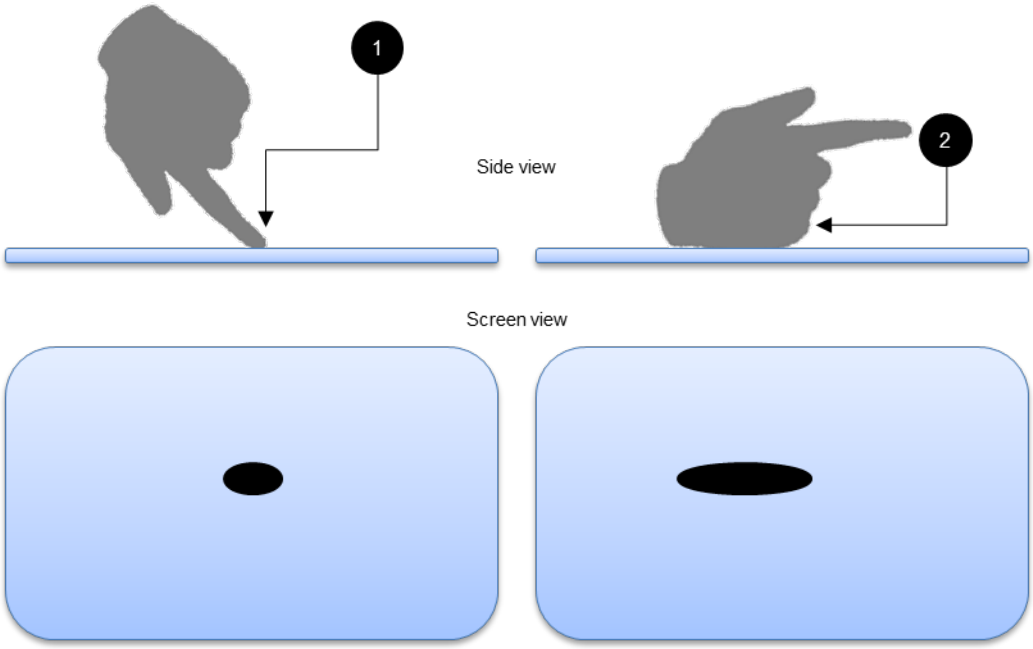


Figure 8.1 As the finger pushes on the touchscreen, the pad of the finger flattens #1. Touchscreens can detect this and report the size of a contact. The contact area increases as the finger pressure increases. Alternate poses such as the side of a hand #2 can be detected in this way.

Figure 8.1 and caption should be in sidebar

Besides pressure and hand pose, the size of the touch can also be used for drawing and displaying auras that provide visual feedback of a finger's detection.

To help you understand when you should use the raw touch API instead of the manipulations API, let's discuss the differences between raw touch and manipulations.

8.1.2 When to use the raw touch API

Raw touch information is most useful when you need to react to individual touches. In contrast, the manipulation API is appropriate when you really want to know about how the touches are changing overall rather than individual touches. You can use both together if necessary, so we're not really talking about a critical either-or choice. In fact, some scenarios require using parts of both APIs.

The choice between raw touch and manipulations comes down to which one lets you write less code to do what you need to do. Table 8.2 shows some programming scenarios and which API you may need to use.

Table 8.2 The raw touch API and manipulation API are useful in different scenarios. Depending upon the task, it may be better to use one, the other, or both.

Scenario	Preferred API	Reason
Water simulation responds to individual fingers	Raw touch	Raw touch events lets you change the water as each finger touches and moves
Count how many fingers are touching a UIElement	Raw touch	Raw touch gives you access to individual touch points and properties for how many points are over a UIElement
Visualize finger touches for touch feedback (auras)	Raw touch	Raw touch gives you the position and sizes of individual touches
Finger paint with variable width strokes	Raw touch	Raw touch gives you the position and size of individual touches
Pan, rotate, or scale a UIElement	Manipulation	Manipulations tell you the pan, rotate, or scale motion of all fingers in aggregate
Track continuous linear, rotation, or scale motion	Manipulation	Tracking of motion is most easily done with manipulations
Recognize a gesture based upon velocity threshold	Manipulation	Manipulations provide velocity information and watching for a threshold is simple
Simulate inertia and friction after an interaction	Manipulation	The manipulation API has an integrated inertia feature
Recognize a gesture based upon timing and count of individual fingers	Raw touch	Raw touch events lets you track the timing and number of fingers
Pan a UIElement with differently based upon the number of fingers in use*	Both	Manipulations would be used to track the motion of the fingers, and raw touch would be used to determine how many fingers were used

* I use this example to show what is possible, but changing manipulation modes based upon the number of fingers is not an intuitive behavior and should only be used in special circumstances

In chapter 3 we discussed the difference between gestures and manipulations. You may wonder whether the manipulations API should be used whenever you want a manipulation. In might seem confusing at first, but really you can implement manipulations and gestures with either the raw touch or the

manipulations API. We will see examples in this chapter of implementing both gestures and manipulations with the raw touch API. First, let's learn what the API itself gives us.

8.1.3 The raw touch API

The raw touch API is composed of a small number of events, properties, and methods that are integrated into the presentation framework, as well as a few classes that provide the actual data. When I say integrated into the presentation framework, I really mean it. The raw touch features we are about to talk about are available for any UIElement, UIElement3D, or ContentElement. Since almost everything in WPF derives from one of these classes, everything you can see in WPF can use touch.

> **NOTE**
>
> In the interest of brevity, whenever I talk about an event, property, or method that is part of an element or the UIElement class, I will skip mentioning UIElement3D and ContentElement. You can safely assume that the features with UIElement are also available on the other two.

Table 8.3 summarizes the different components of the raw touch API.

Table 8.3 The raw touch API is composed of events, properties, and methods that are available on any UIElement.

Type	Description	API

Events	Fired when fingers touch the screen, move, or change state	`EventHandler<TouchEventArgs> GotTouchCapture`
		`EventHandler<TouchEventArgs> LostTouchCapture`
		`EventHandler<TouchEventArgs> TouchDown`
		`EventHandler<TouchEventArgs> TouchEnter`
		`EventHandler<TouchEventArgs> TouchLeave`
		`EventHandler<TouchEventArgs> TouchMove`
		`EventHandler<TouchEventArgs> TouchUp`
		`EventHandler<TouchEventArgs> PreviewTouchDown`
		`EventHandler<TouchEventArgs> PreviewTouchMove`
		`EventHandler<TouchEventArgs> PreviewTouchUp`

Properties	Exposes the current state of all touches over or captured to an element	`bool AreAnyTouchesCaptured`
		`bool AreAnyTouchesCapturedWithin`
		`bool AreAnyTouchesDirectlyOver`
		`bool AreAnyTouchesOver`
		`IEnumerable<TouchDevice> TouchesCaptured`
		`IEnumerable<TouchDevice> TouchesCapturedWithin`
		`IEnumerable<TouchDevice> TouchesDirectlyOver`
		`IEnumerable<TouchDevice> TouchesOver`

Methods	Enables capture or release of TouchDevices	`bool CaptureTouch(TouchDevice device)`
		`bool ReleaseTouchCapture(TouchDevice device)`
		`void ReleaseAllTouchCaptures()`

The events notify us when touches are changing. The properties allow us to get access to all of the TouchDevices interacting with a UIElement without manually tracking them. The methods allow us to control the capture status of a TouchDevice. In the next few sections we will be learning the specifics of how to make use of these features. TouchDevice is one of several supporting classes added to the framework, detailed in table 8.4.

Table 8.4 The raw touch API also includes several supporting classes.

Class	Description
TouchEventArgs	The common EventArgs used for all touch event handlers
TouchDevice	Represents a single instance of a finger touch
TouchPoint	Represents a snapshot of a TouchDevice's current status
Touch	Exposes the FrameReported event for Silverlight compatibility.

In most cases, you will not need to instantiate these classes directly. Rather, they are instantiated by the framework and provided to you pre-populated with information. For example, the TouchesOver property exposes a list of TouchDevices that are over that particular UIElement.

The Touch class and FrameReported

The Touch class is static and exposes a single event: FrameReported. This event reports raw touch frames, or events, across the entire application without hit testing. This approach is very rudimentary and is only available in WPF to enable source compatibility with Silverlight. Unless you are porting Silverlight touch code to WPF, you will almost never need to use this class or event. The rest of the raw touch API provides a higher level of abstraction that is much more useful than the raw data from FrameReported.

When you are working in WPF, there is no need to mess with it, so I will not cover this class or event here. In part 3, we will revisit Touch.FrameReported because this is the only touch event that Silverlight exposes.

The API is not that large so in this chapter we will walk through the events, properties, methods, and classes that you can use for accessing raw touch data. In section 8.3 and the rest of this chapter, we'll practice using the API with coding walkthroughs that illustrate how to use the API in different scenarios. Since you'll be using the events the most, let's start with those.

8.2 Raw touch events

There are ten touch events integrated into UIElement, but some of them are the "preview" events for other routed events. You can subscribe to these events if you want to know when touch events occur on a particular UIElement. Figure 8.2 illustrates the basic touch events and their sequence.

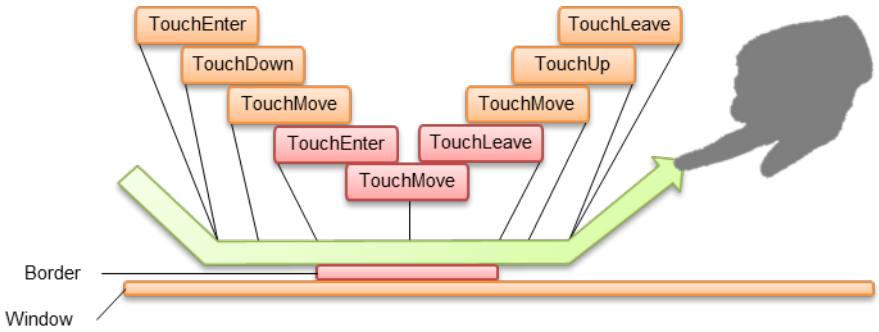


Figure 8.2 As a finger contacts the visual elements, slides across them, and lifts from the screen, WPF will generate a series of touch events. In this illustration, the window would receive TouchEnter, TouchDown, TouchMove, TouchUp, and TouchLeave events. The smaller Border would receive TouchEnter, TouchMove, and TouchLeave events.

The TouchDown and TouchUp events are generated when the finger touches or lifts from the touch surface, but TouchEnter and TouchLeave events are also generated in the same case. If you want to

detect when a finger either touches down within a UIElement or slides into it from elsewhere, you can just use TouchEnter and it will handle both of those cases.

The touch events are similar to the similarly named mouse events. Table 8.5 describes all of the touch events and shows the equivalent mouse events. Note that there is no relationship between the mouse and touch events. They just follow the same API pattern. The two primary differences between the mouse and touch events are that touch events of course respond to as many touches as the hardware supports and the touch events give us slightly different information than the mouse events, which we will see in the next section.

Table 8.5 The UIElement class has the several events that you can use to be notified when particular touch events occur. All of these events occur relative to the UIElement they are attached to. The mouse events are listed to help you understand the parallels between the mouse and touch API.

Touch Event	Mouse Equivalent	Description
TouchDown	MouseDown	Occurs when a finger first contacts the touchscreen over an element
PreviewTouchDown	PreviewMouseDown	TouchDown event routed from root element to source element
TouchMove	MouseMove	Occurs when a down finger moves over an element
PreviewTouchMove	PreviewMouseMove	TouchMove event routed from root element to source element
TouchUp	MouseUp	Occurs when a down finger is lifted from the touchscreen over an element
PreviewTouchUp	PreviewMouseUp	TouchUp event routed from root element to source element
TouchEnter	MouseEnter	Occurs when a down finger enters the boundary of the element as well as just before a TouchDown event
TouchLeave	MouseLeave	Occurs when a down finger leaves the boundary of the element as well as just after a TouchLeave event
GotTouchCapture*	GotMouseCapture	Occurs when a touch is captured to an element
LostTouchCapture*	LostMouseCapture	Occurs when a captured touch is released from an element

*We will discuss touch capture in detail in section 8.4.

Each of these events has the same signature. The definition of the event looks like this:

```
public event EventHandler<TouchEventArgs> TouchDown;
```

You can use standard techniques to subscribe to this event either in code:

```
border.TouchDown += new EventHandler<TouchEventArgs>(border_TouchDown);
```

or in XAML:

```
<Border Name="border" TouchDown="border_TouchDown">
</Border>
```

The TouchDown or any of these events you just need to have an event handler with this signature:

```
void border_TouchDown(object sender, TouchEventArgs e)
{
}
```

The event handler uses the TouchEventArgs class to give you access to the touch information. The TouchEventArgs class is derived from InputEventArgs, so some of the properties and methods may look familiar to you. Table 8.6 lists the TouchEventArgs members that are unique to this class.

Table 8.6 The TouchEventArgs class provides a few different methods and properties that will be useful for accessing the data from table 8.1. This table does not include inherited members.

Name	Type	Description
GetIntermediateTouchPoints	Method	Returns all touch points since the last touch event
GetTouchPoint	Method	Returns the current touch point
TouchDevice	Property	Gets the touch device that generated the event

The following code snippet shows an example of getting the important classes from the TouchEventArgs.

```
private void border_TouchDown(object sender, TouchEventArgs e)
{
    TouchDevice device = e.TouchDevice;
    TouchPoint point = e.GetTouchPoint(border);
    TouchPointCollection pointCollection =
       e.GetIntermediateTouchPoints(border);
}
```

You need the TouchDevice and TouchPoint classes to figure out what is going on with the touch events. Let's discuss these two classes and the role they play.

8.2.2 TouchDevice and TouchPoint

A `TouchDevice` instance represents a single finger contact that can generate many touch events. A `TouchPoint` instance represents the state of a contact at a particular time. You use the `TouchPoint` class to access data such as the current state, position, or size of a contact. In contrast, you can use the `TouchDevice` to keep track of the `TouchPoints` generated by a single finger across several touch events. You also need `TouchDevice` to capture a specific contact. Capture will be discussed in section 8.4.

Tracking individual fingers may be important for many different reasons. If you are tracing out lines with multiple fingers, for example, you may be keeping track of the previous position and drawing a line to the new position in `TouchMove`. In this case it would be important to tell which finger and which previous position a `TouchMove` event corresponds to. Figure 8.3 illustrates how the touch events may jump back and forth between two fingers but the TouchDevice helps keep things organized.

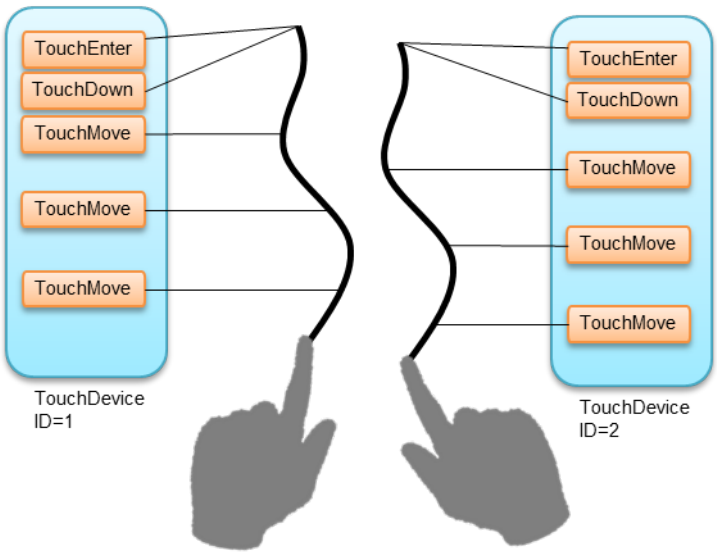


Figure 8.3 As two fingers trace separate paths, a series of touch events is raised. Notice that the `TouchMove` events alternate from one finger to another. We can sort them out by checking the ID value of the `TouchDevice`. In this illustration, ID 1 was assigned to the left finger and ID 2 was assigned to the right finger, which touched down slightly after

the left finger. If we drew line segments between consecutive `TouchMove` events without checking the ID, it would create a single zig-zag instead of two smooth lines.

Although there are two methods to get TouchPoints, `GetIntermediateTouchPoints` and `GetTouchPoint`, the majority of the time you will only need the `GetTouchPoint` variety. WPF rate limits the event calls for performance reasons so if the fingers are moving particularly quickly or you are using hardware with a high-frequency report rate, you may not see all of the `TouchPoints`. In most cases this is fine, but if you want to access all of the points for a precise task you can call `GetIntermediateTouchPoints`.

Both `GetTouchPoint` and `GetIntermediateTouchPoints` take an `IInputElement` as a parameter. The `IInputElement` determines the frame-of-reference for the `TouchPoint`'s position. This is useful because if you are going to use the positions in the context of a particular control or element, you can pass that element to GetTouchPoint and the origins will line up. If you pass null, the `TouchPoint` position will be relative to the WPF window. Figure 8.4 illustrates example of using different parameters for `GetTouchPoint` within the same event.

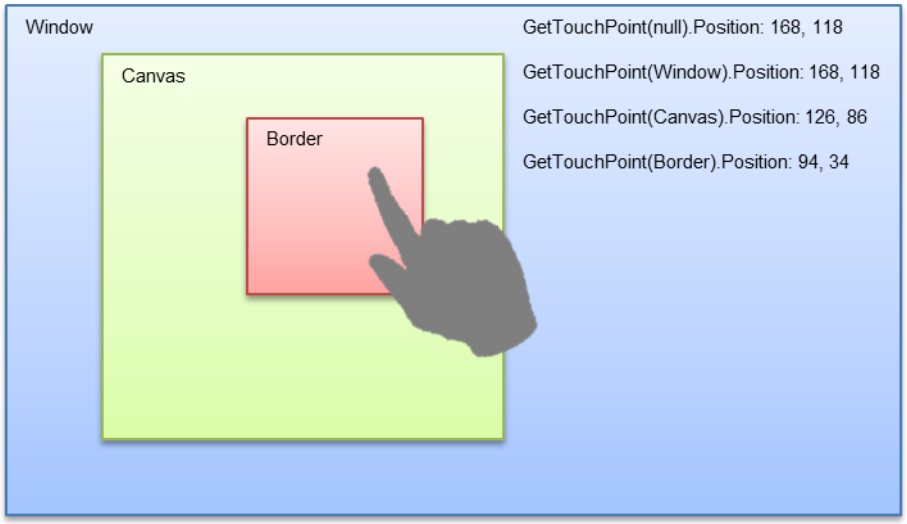


Figure 8.4 You can get the position of a TouchPoint in different frames of reference by passing different parameters to GetTouchPoint. Here the same TouchPoint is reported relative to the Window, Canvas, and Border. Passing null will return the TouchPoint relative to the WPF window.

The TouchPoint class gives us position, action, size, and bounds. Table 8.7 details the properties.

Table 8.7 The TouchPoint contains the data about the current status of a touch contact.

Property	Type	Description
Action	TouchAction	The action that the event represents. Possible values of Down, Move, or Up.
Bounds	Rect	The relative bounds of the area of the finger in contact with the screen
Position	Point	The relative center of the finger contact
Size	Size	The size of the area of the finger in contact with the screen
TouchDevice	TouchDevice	The TouchDevice that generated this event

These events, `TouchDevice`, and `TouchPoint` are only part of the raw touch API, but before we get to the rest of the API, let's make a basic program to test out what we have learned already.

8.3 Testing Basic Touch Events

Let's create a simple program, called TouchTraces, that will show how to use the basic touch events to get information from a `TouchPoint` and also demonstrate how to track multiple fingers by using the `TouchDevice`. We're going to make this program do two things: display real-time information about each `TouchPoint` and drawing traces as the fingers move. In the end you will end up with something that looks like figure 8.5.

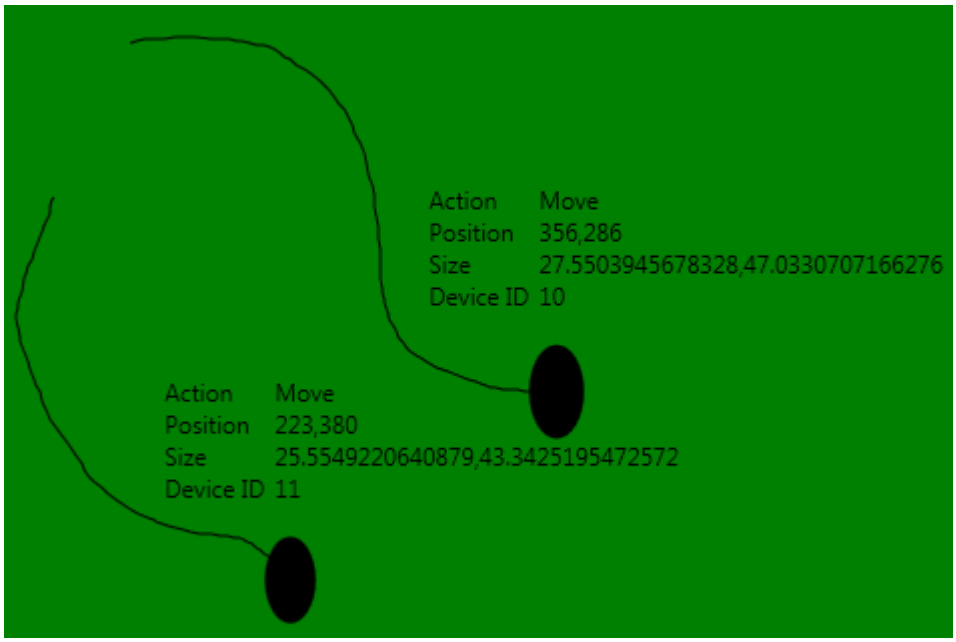


Figure 8.5 The TouchTrace program displays information about each `TouchPoint` and displays traces as the fingers move.

To start out, launch Visual Studio 2010 and start a new WPF Application and call it TouchTraces. Open MainWindow.xaml and replace the contents with this:

```
<Window x:Class="TouchTraces.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow"
        Width="800"
        Height="600"
        Top="10"
        Left="10">
    <Grid>
        <Canvas Name="mainCanvas"
                Background="Green"
                Width="800"
                Height="600"
                TouchDown="mainCanvas_TouchDown"
                TouchMove="mainCanvas_TouchMove"
                TouchUp="mainCanvas_TouchUp">
        </Canvas>

    </Grid>
</Window>
```

Before we create the event handlers for TouchDown, TouchMove, and TouchUp, we need to create a user control to display TouchPoint information. Right click on the TouchTraces project, click Add then click User Control. Name the user control TouchPointData.xaml. Open the file TouchPointData.xaml and replace the content with listing 8.1. This will create a control that displays information about a TouchPoint and a representative ellipse. Two TouchPointData controls are shown in figure 8.5.

Listing 8.1 TouchPointData.xaml will be bound to a TouchPoint to display data

```
<UserControl x:Class="TouchTraces.TouchPointData"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Canvas>
        <Grid Canvas.Top="-80"
              Canvas.Left="-50">
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <TextBlock Grid.Column="0"
                       Grid.Row="0"
                       Text="Action" />
            <TextBlock Grid.Column="1"
                       Grid.Row="0"
                       Text="{Binding Action}" />

            <TextBlock Grid.Column="0"
                       Grid.Row="1"
                       Text="Position" />
            <TextBlock Grid.Column="1"
                       Grid.Row="1"
                       Text="{Binding Position}" />

            <TextBlock Grid.Column="0"
                       Grid.Row="2"
                       Text="Size" />
            <TextBlock Grid.Column="1"
                       Grid.Row="2"
                       Text="{Binding Size}" />

            <TextBlock Grid.Column="0"
                       Grid.Row="3"
                       Text="Device ID"
                       Margin="0,0,5,0" />
            <TextBlock Grid.Column="1"
                       Grid.Row="3"
                       Text="{Binding TouchDevice.Id}" />
        </Grid>

        <Ellipse Width="{Binding Size.Width}"
                 Height="{Binding Size.Height}"
                 Fill="Black"
                 HorizontalAlignment="Center" />
    </Canvas>
</UserControl>
```

This XAML creates a simple grid with labels for some of the important information that the TouchPoint gives us access to. It uses standard data binding to display the information. In order for this to work, we must set up the DataContext in the code behind. Open TouchPointData.xaml.cs and add the following code within the `TouchPointData` class:

```
private TouchPoint _touchPoint;
public TouchPoint TouchPoint
{
    get { return this._touchPoint; }
    set
    {
        this._touchPoint = value;
        this.DataContext = _touchPoint;
    }
}
```

This property lets us set the TouchPoint that this TouchPointData control will display. When you set the property it will update the DataContext, which will allow all of the data binding we set up in the XAML to work.

Now what we need to do is add some code to the touch event handlers to add, update, and delete TouchPointData controls. Open MainWindows.xaml.cs and add the code in listing 8.2 to the MainWindow class. This code will work for a single finger but will misbehave with multiple fingers. Can you figure out why?

Listing 8.2 Display and update TouchPoint data

```
TouchPointData touchData;

private void UpdateTouchData(TouchEventArgs e)
{
    touchData.TouchPoint = e.GetTouchPoint(null);           #A
    Canvas.SetLeft(touchData,                               #A
                   touchData.TouchPoint.Position.X -        #A
                   touchData.TouchPoint.Size.Width / 2);    #A
    Canvas.SetTop(touchData,                                #A
                  touchData.TouchPoint.Position.Y -         #A
                  touchData.TouchPoint.Size.Height / 2);    #A
}

private void mainCanvas_TouchDown(object sender, TouchEventArgs e)
{
    if (touchData != null)
        mainCanvas.Children.Clear();

    touchData = new TouchPointData();                           #B
    mainCanvas.Children.Add(touchData);                         #B

    UpdateTouchData(e);
}

private void mainCanvas_TouchMove(object sender, TouchEventArgs e)
{
    UpdateTouchData(e);
}

private void mainCanvas_TouchUp(object sender, TouchEventArgs e)
{
    mainCanvas.Children.Clear();                                #C
}
```

#A Updates TouchPointData and Canvas position
#B Creates new TouchPointData, adds to Canvas
#C Clears Canvas when finger up

If you run the program now, it will display the TouchPointData control when you touch it and the control will follow your finger around. If your touch hardware reports TouchPoint sizes, then the ellipse will change shape to match the approximate finger area in contact with the screen.

There is a problem, though, because if you touch multiple fingers at once the single TouchPointData control will flicker between each finger. This illustrates a core issue when dealing with multi-point input compared to single-point cursors. With multi-input, we can get a stream of events but to effectively make sense of them we have to correlate which events correspond to a particular finger. This is where we can use the TouchDevice.

As we discussed earlier, a TouchDevice helps us relate all the events that were generated by a single finger. Let's modify the code we just entered to take advantage of this. Delete the code from listing 8.2 and replace it with the updated version in listing 8.3. This code will work well with multiple fingers and will also draw a line trace of the finger movement.

Listing 8.3 Track multiple fingers with TouchDevice

```
Dictionary<TouchDevice, TouchPointData> touches =        #1
    new Dictionary<TouchDevice, TouchPointData>();       #1
Dictionary<TouchDevice, Point> lastPoints =              #1
    new Dictionary<TouchDevice, Point>();                #1


private void UpdateTouchData(TouchEventArgs e)
{
    TouchPointData data;

    if (!touches.TryGetValue(e.TouchDevice, out data))   #A
    {
        data = new TouchPointData();                     #B
        touches.Add(e.TouchDevice, data);        #B
        mainCanvas.Children.Add(data);           #B
    }

    data.TouchPoint = e.GetTouchPoint(null);

    Canvas.SetLeft(data,
                   data.TouchPoint.Position.X -
                   data.TouchPoint.Size.Width / 2);
    Canvas.SetTop(data,
                  data.TouchPoint.Position.Y -
                  data.TouchPoint.Size.Height / 2);
}
```

#A If no TouchPointData associated with TouchDevice
#B Create new TouchPointData, add to canvas

Cueballs in code and text

This new code so far has changed how we are storing the current TouchPointData references. Instead of holding and updating a single TouchPointData reference, we are using a Dictionary to associate TouchDevices to TouchPointData #1. We also use a Dictionary to associate TouchDevices to the last reported TouchPoint position of that TouchDevice. These are used in the UpdateTouchData method above to update the appropriate TouchPointData instance depending upon which TouchDevice generated the event.

Listing 8.3 (cont.)

```
private void mainCanvas_TouchDown(object sender, TouchEventArgs e)
{
    UpdateTouchData(e);

    lastPoints[e.TouchDevice] =                                  #C
           e.GetTouchPoint(null).Position;                       #C
}

private void mainCanvas_TouchMove(object sender, TouchEventArgs e)
{
    UpdateTouchData(e);

    Point point = e.GetTouchPoint(null).Position;                #D
    Point lastPoint = lastPoints[e.TouchDevice];                 #D
                                                                 #D
    Line line = new Line();                                      #D
    line.Stroke = new SolidColorBrush(Colors.Black);             #D
    line.StrokeThickness = 1;                                    #D
    line.X1 = lastPoint.X;                                       #D
    line.Y1 = lastPoint.Y;                                       #D
    line.X2 = point.X;                                           #D
    line.Y2 = point.Y;                                           #D
    line.Tag = e.TouchDevice;                                    #D
                                                                 #D
    mainCanvas.Children.Add(line);                               #D
```

```
        lastPoints[e.TouchDevice] = point;
    }

    private void mainCanvas_TouchUp(object sender, TouchEventArgs e)
    {
        TouchPointData data;

        if (touches.TryGetValue(e.TouchDevice, out data))           #E
        {                                                           #E
            touches.Remove(e.TouchDevice);                          #E
                                                                    #E
            if (mainCanvas.Children.Contains(data))                 #E
            {                                                       #E
                mainCanvas.Children.Remove(data);                   #E
            }
        }

        List<Line> linesToDelete = new List<Line>();

        foreach (UIElement element in mainCanvas.Children)
        {
            Line line = element as Line;
            if (line != null &&                                             #F
                line.Tag == e.TouchDevice)                                  #F
            {                                                               #F
                linesToDelete.Add(line);                                    #F
            }
        }

        foreach (Line line in linesToDelete)
        {
            mainCanvas.Children.Remove(line);                               #G
        }
    }
```

#C Save last position of touch
#D Add line segments
#E Remove TouchPointData for current TouchDevice
#F Find lines tagged with TouchDevice
#G Remove marked line segments

These event handlers call the UpdateTouchData method and also draw line segments from the previous position stored in the lastPoints Dictionary and then update the stored position. The mainCanvas_TouchUp also removes the TouchPointData and any line segments associated with the TouchDevice that was just lifted.

Now this code will work properly and will produce the result from figure 8.5. The code will create lines that follow the path of your fingers. When you lift a finger, the line created from that finger will disappear. Go ahead and play with it for a few minutes.

> **NOTE**
>
> With TouchTraces, if you draw the lines for too long without lifting your fingers it will start lagging. This is due to the large number of line segments it creates and the naïve implementation. A better implementation would limit the number of line segments and remove the oldest part of the trace, but that is left as an exercise for the reader. The code is presented as-is so as to focus on learning the raw touch API rather than creating the most efficient program.

Now that we have learned about the touch events, let's continue to discuss the rest of the touch API. There are only a couple of methods and properties that we need to discuss and they primarily deal with capturing touch devices.

8.4 Capturing touching devices

The concept of capturing touches is critical to understanding how the touch system works. Mouse input can also be captured, but touch capture is more complex since we are talking about multiple points and multiple touch devices.

All of the touch events are routed events. By default, WPF routes touch events to the visual element that the touch is directly over. This means if you touch down over one element, then move your finger out of that element to another element, the first one will no longer receive any touch events. In some cases, which we will discuss, it may be important that the original element continues to receive all of the touch events from that touch device, even if the finger moves outside of the boundary of the element. Figure 8.6 explains the difference between captured and non-capture touch events.

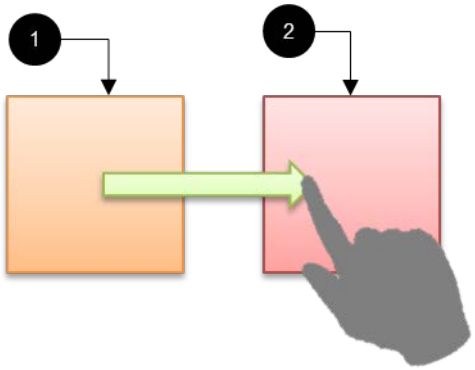


Figure 8.6 A finger is placed down over rectangle #1, slid to rectangle #2, then released. Without capture, rectangle #1 would receive TouchEnter, TouchDown, several TouchMoves, and TouchLeave events, while rectangle #2 would receive TouchEnter, several TouchMoves, TouchUp, and TouchLeave events. If rectangle #1 captured the TouchDevice when it was placed down, then it would have received all touch events regardless of the position of the finger.

Capturing the touch device is preferred any time the user expects to continue to interact with a particular visual element regardless of finger position. This might include controls such as a button or slider, as well as objects that involve panning, zooming, and rotation. Typically pan, zoom, and rotation is handled by using the manipulation API, covered in chapter 9. Manipulations automatically capture touch devices.

You may want to manually capture touch devices if you are implementing a control or a gesture. We discussed the SurfaceButton in chapter 5 and how it will only trigger a Click event once all of the fingers that pressed the button have been lifted. Internally it tracks this by capturing all of the touch devices. If you need to create a control, you may want to have a similar behavior.

Controlling the capture of a touch device is very easy. Let's discuss the methods that you will need.

8.4.1 Touch capture methods and events

WPF provides three touch-specific UIElement methods in the raw touch API. Table 8.8 introduces these methods.

Table 8.8 WPF has three raw touch methods, available on UIElement, UIElement3D, and ContentElement. These methods help you manage the capture of touch devices.

Method	Description
bool CaptureTouch(TouchDevice device)	Captures the specified TouchDevice to an element
bool ReleaseTouchCapture(TouchDevice device)	Releases the capture on a TouchDevice from an element
void ReleaseAllTouchCaptures()	Releases all captured TouchDevices from an element

The following code snippet shows how you might capture a touch device in the TouchDown event handler of a UIElement. This code ensures that all future touch events for the current TouchDevice are sent directly to the border element.

```
void border_TouchDown(object sender, TouchEventArgs e)
{
    border.CaptureTouch(e.TouchDevice);
    e.Handled = true;
}
```

We set the event to handled to ensure that other element farther up in the visual tree don't get the TouchDown event and possibly steal the capture. On the other hand, if some element inside this border element had already captured the device but did not set the event to be handled, this code would steal the capture.

We may want to perform certain actions when a touch is captured or capture is lost. This might include changing a visual or updating an internal data structure that tracks touch devices. Since any code can capture a touch to any other UIElement, we may not have control over the capture logic. Luckily, there are two events that will fire when a capture is gained or lost.

When a touch is captured to an element, the GotTouchCapture event is raised. If an element loses a capture, the LostTouchCapture event is raised. Both of these events use the TouchEventArgs parameter like the other touch event handlers. The LostTouchCapture event is called in three cases:

- When ReleaseTouchCapture or ReleaseAllTouchCaptures is called on an element
- When another element captures the same TouchDevice ("stealing" the capture)
- When the finger is lifted and the TouchDevice is deactivated

It is important to keep in mind that with multi-touch screens, many fingers may be interacting with the interface at the same time and thus there may be several different simultaneous captures going on. A single UIElement may have multiple touch devices captured at once. Several different UIElements may have different touch devices captured. A touch device may only be captured by zero or one UIElement at a time, however.

In some cases you may be interested in figuring out how many fingers are over or captured to a particular element. You could set up the touch events and track this information manually, but that would be somewhat tedious and error prone. Helpfully, WPF provides several properties on UIElement that gives us access to this information. These properties are the last piece of the raw touch API that we need to learn.

8.5 Touch properties

While raw touch events notify you any time a touch contact changes, the eight raw touch properties give you access to all of the TouchDevices that are over or captured to a UIElement any time you need to check. This saves you from having to track the TouchDevices yourself. In this section I will show you what these events are, when you might want to use each property, and then show you how to use them. First, let's review the actual properties, shown in table 7.9.

Table 7.9 The raw touch API exposes the following properties on all UIElement

Property	Description
`bool AreAnyTouchesCaptured`	If touches are captured to this element
`bool AreAnyTouchesCapturedWithin`	If touches are captured to this element or child elements
`bool AreAnyTouchesDirectlyOver`	If touches are over this element
`bool AreAnyTouchesOver`	If touches are over this element or child elements
`IEnumerable<TouchDevice> TouchesCaptured`	All touches captured to this element
`IEnumerable<TouchDevice> TouchesCapturedWithin`	All touches captured to this element

	or child elements
`IEnumerable<TouchDevice> TouchesDirectlyOver`	All touches over this element
`IEnumerable<TouchDevice> TouchesOver`	All touches over this element or child elements

There are only eight properties, and they are pretty easy to use. If you have a UIElement that you need to check the TouchDevices on, simply access any of these properties. For example, suppose you have a Border named "borderMain" and a TouchMove event handler on that Border. In the TouchMove event handler, you want to activate or deactivate a visual effect depending upon if there are any touches over the element. In addition, you want to iterate through these touches and perform some custom hit testing and possibly capture them. You might accomplish that using the following code:

```
private void borderMain_TouchMove(object sender, TouchEventArgs e)
{
    if (borderMain.AreAnyTouchesOver)
        ActivateVisual();
    else
        DeactivateVisual();

    foreach (TouchDevice device in borderMain.TouchesOver)
    {
        if (CustomHitTest(device))
            borderMain.CaptureTouch(device);
        else
            borderMain.ReleaseTouch(device);
    }
}
```

Even though there are only eight properties, they are fairly similar to each other. It may be confusing at first to figure out which property you need. Let's talk about when you should use each property.

8.5.1 When to use which touch property

These touch properties have a very simple symmetry. There are really two sets of four properties, and the "AreAny" variants are a shortcut to checking whether the Count() of the IEnumerable properties is zero. Figure 8.7 describes the four properties within each set.

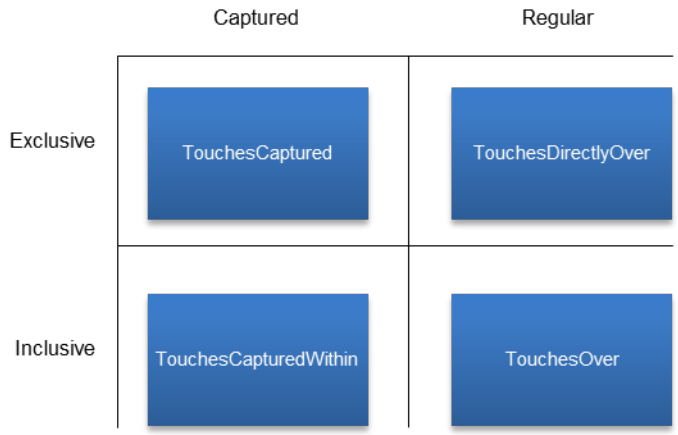


Figure 8.7 The raw touch API properties fall into two categories: whether they return regular or captured touches, and whether touches exclude or include child elements. The exclusive variants only return touches captured or over the specific UIElement that the property is attached to. This same chart applies to the "AreAny" set of properties.

You may wonder why we need to have the exclusive and inclusive variants, such as TouchesOver versus TouchesDirectlyOver. The reason is that most of the time, such as when you have custom controls or other composed visual elements, you will want to operate on an entire sub-tree of elements.

Suppose you are creating a content control with several nested visual elements that make up the look and feel of the control. In most cases, if you need to get the touches interacting with the element you will want to use TouchesOver property. This is because the touch might be directly over a child element, but not the root element. TouchesDirectlyOver would only return the touches directly over a single element. You might need to get these touches so that you can detect a gesture or update a visual depending upon the number and position of the touches.

On the other hand, if you are creating an items control that lays out the children in a specific way, such as a custom ScatterView control, you may in fact want to operate on touches that are just over the background visual element but not over the child ScatterViewItems. You might want to do this if you want to create background touch visualizations, such as auras or water, but only when the touches are over the background.

The same applies to the capture variety of properties, with one twist. In most cases, you will want to use the exclusive variety of capture, TouchesCaptured. This is because when a TouchDevice is captured to an element, it is usually captured to the root element of a control. You would want to use TouchesCaptured to get access to the touches intentionally captured to a specific control. As an example, when you touch a SurfaceButton it captures your touches to the entire button.

Sometimes you might have a child control, such as a Slider inside of a ScrollViewer, that also captures the touch. TouchCapturedWithin on the outer ScrollViewer would return the touches captured to the ScrollViewer as well as Slider and any other controls. It would be rare for you to need to get all of those touches.

It may seem a bit overwhelming at this point to figure out what all of this means. The best way to figure out what is going on is to just play with it. To help out, in the next section I will introduce a relatively simple program designed for you to play around and get to know capture and these touch properties.

8.5.2 Raw touch playground

The raw touch playground sample application is a very simple program with a single purpose: enable you to touch and explore the different raw touch properties, element composition, and capture. The program, shown in figure 8.8, consists of an outer border and an inner border.

Figure 8.8 The raw touch playground sample application lets you explore the raw touch properties and capture. It displays the number of touches in the raw touch properties and allows you to enable or disable capturing touches.

Each border displays the number of touches in the raw touch properties: TouchesOver, TouchesDirectlyOver, TouchesCaptures, and TouchesCapturedWithin. You can also toggle whether the borders will capture touch points in the TouchDown event.

Try running the program and then touching various parts of the screen, both inside and outside the borders. Drag your finger or fingers from the outer border to the inner border and observe how the touch counts change. Without capture, only the TouchesOver and TouchesDirectlyOver properties will change.

Once you get a feel for what is going on, try toggling the checkboxes to turn on capture. With capture on, the border will capture any TouchDevice that touches down in that border. Once the TouchDevice is captured, events will be routed to the capturing border no matter where the touch moves. Try capturing a touch in the inner border and then drag it to the outer border. Notice the counts do not change at all. The inner border still receives the events and the touch properties reflect this as well.

One additional thing to notice: in order to make it apparent when a touch is captured but outside of the border's boundary, I have used a touch visualization adapter. Figure 7.9 shows the visual tether created by the touch visualization adapter.

Figure 8.9 A touch visualization adapter is used to enable the visual tethers between the inner border and the touch visualizations. Also note that even though the four touches are outside of the inner border, they are captured to the inner border so the properties displayed act as if the touches were still inside the border.

The following code snippet shows how the touch visualization adapter was implemented. I chose the TouchVisualizerRectangleAdapter since we are using a rectangular shape. I gave it the same corner radius as the border so the tethers would also appear to connect nicely around the border corners.

```
<Border Name="bdrInner"
        CornerRadius="20"
        [...]>
    <s:TouchVisualizer.Adapter>
        <s:TouchVisualizerRectangleAdapter RadiusX="20"
                                           RadiusY="20" />
    </s:TouchVisualizer.Adapter>
    [...]
</Border>
```

If you're interested in how the rest of the code works, you can take a look at the provided source. I kept this program as simple as possible so you can easily digest what is going on.

In this section, we learned about the touch properties. This was the last new part of the raw touch API that we need to learn and use, but there is one more important concept that you need to know to fully understand what is going on with touch: mouse promotion. Once we get through this next section, we will take a look at how raw touch can be applied in different scenarios in section 8.7 and then we will wrap this chapter up.

8.6 Mouse promotion

When Microsoft was planning WPF 4 and the Touch API, one of the considerations was how will touch work with applications that are not specifically designed with the Touch API. There are many existing GUI WPF applications that users with touch screens should still be able to interact with using touch, even though it may not be the best user experience.

The Touch API includes an entirely different touch event model from the mouse or stylus events. Since that is the case, when the user touches the screen and the framework generates touch events, how will those events activate a control that responds to only mouse events? The solution is mouse promotion.

Routed events review

WPF has a concept called routed events. In case you are not familiar with routed events, here is a little refresher that will be useful to fully comprehend how and why mouse promotion works.

Touch events are routed events and they are routed to the source element. The source element is the visual element that the touch is over, determined by hit testing, or the element it is captured to if the `TouchDevice` is captured. The "preview" event, such as `PreviewTouchDown`, is first raised at the root visual. If there is no event handler or the event handler does not set `e.Handled = true`, then `PreviewTouchDown` is raised on the child element that leads to the source element. This is called tunneling and the process repeats until `PreviewTouchDown` is raised on the source element.

If none of the elements handle the preview event, then the regular `TouchDown` event is raised on the source element first. If this element does not handle the event, then `TouchDown` is raised on the parent element. This is called bubbling and repeats until the event is raised on the root element.

Mouse promotion enables unhandled touch events to automatically be transformed into mouse events. In this section we will discuss the primary touch point, how it relates to mouse promotion, and when and why mouse promotion occurs.

THE PRIMARY TOUCH POINT

Even though in multi-touch, each touch is no different than the others, for the purposes of mouse promotion there is one touch that is special: the primary touch point. There are always either zero or one primary touch points and the primary touch point is assigned to the first touch on a screen only when there were previously no touches.

Consider this scenario: there are no touches on the screen. The user touches and holds with finger #1. This finger is the primary touch point. The user then touches a second finger to the screen. Finger #1 is still the primary touch point, even as the two fingers move around, and even if finger #2 is lifted or placed again. If fingers #1 and #2 are in contact with the screen and the user lifts the primary touch point (finger #1), then finger #2 does not become the primary touch point. Even if the user, while still holding finger #2 down, touches another finger, that does not become the primary touch point either. A new primary touch point will not be assigned until all fingers have lifted and a new finger is touched again.

Windows 7 creates the small cross cursor under the primary touch point, so you can test touching a blank area of the desktop with one or more fingers to see which one gets the little cursor in which circumstances.

The primary touch point is important for this discussion because it is the only TouchDevice that will undergo mouse promotion. Let's talk about when mouse promotion occurs.

PROMOTING TOUCH EVENTS TO MOUSE EVENTS

Mouse promotion occurs if these two criteria are fulfilled:

1. The TouchDevice is the primary touch point
2. None of the PreviewTouchDown or TouchDown events were handled.

In the cases when the primary touch point's TouchDown event bubbles back up to the root element unhandled, WPF will automatically convert the touch event into a mouse event and start PreviewMouseDown routing to the same source element. If you handle any of the touch events, though, mouse promotion will not occur. This is usually not a problem if you are writing a touch application.

It should be noted, again, that because mouse promotion only works for the primary touch point there can only be a single mouse event occurring. This means non-touch applications still only respond to a single touch at a time, but this allows controls such as the native WPF button to respond to mouse events that were originally generated by touch, shown in figure 8.10.

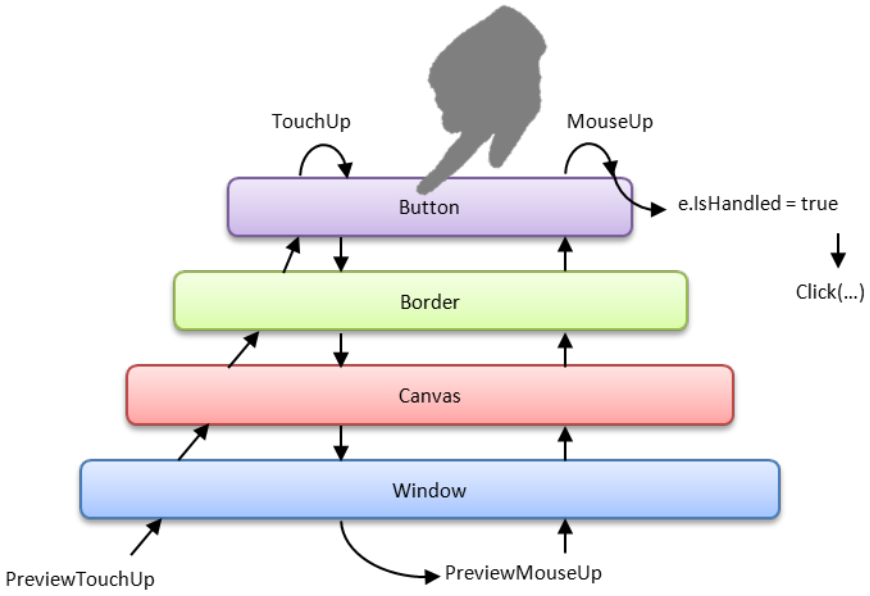


Figure 8.10 A native WPF button responds to touch. The touch events are unhandled, so WPF promotes the events to mouse events. The button responds to the mouse event and triggers a click action.

Mouse promotion makes it possible to use controls designed for the mouse with touch, although it is not an optimal experience. The perfect example of why it is not optimal is this button again. Recall that mouse promotion collapses multi-touch into single mouse events. If touch two fingers to a native WPF button and lift one while holding the other down, the intuitive response would be that the button stays pressed. Instead, due to the subtleties of mouse promotion, a click occurs when the first finger is lifted. This is why Surface SDK and SurfaceButton are useful. Those controls were designed with touch in mind.

We will revisit mouse promotion briefly in chapter 9 to discuss why certain native controls may not work when they are inside of a container with manipulations.

We have now covered all of the necessary raw touch API and concepts. In the next section I will show you a few ways that you can apply the raw touch API.

8.7 Raw touch applied

The raw touch API seems relatively simple, but it can enable some pretty interesting scenarios. In this section I'm going to introduce a few of these scenarios, show a screenshot and provide a reference to the sample project. These projects are available as a part of this book's downloadable sample code at http://manning.com/blake. I won't be going deep into any code descriptions here, but feel free to take a look at the source code, figure it out, and modify it to do something new.

8.7.1 Custom touch visualizations

Although the Surface SDK provides touch visualizations, you may have a need to create custom visualizations. The CustomTouchVisualizations sample project, shown in figure 8.11, shows a simple implementation of using the raw touch API to create your own touch visualizations. This sample requires touch hardware that reports TouchPoint sizes.

Figure 8.11 The CustomTouchVisualizations sample project shows how you can use the raw touch API to create your own custom touch visualizations. Whenever a finger touches the window, an ellipse is created with the position and size of the TouchPoint.

The visualizations are implemented using a class called AuraAdorner. In the Loaded event of your window, you can call:

```
AuraAdorner.AddAuras(this.LayoutRoot);
```

This will automatically subscribe to events and update the adorner layer with ellipses for each of the touches.

You can use the same general technique as shown with custom auras to display pop-up items at each touch contact as well as input for more advanced visualizations such as a water simulation or other ambient deformation.

Responding to each touch visually is an important application of the raw touch API, but it is only one example. Raw touch information is also important for gesture recognition. In the next sample, I'll show how you can use raw touch to recognize a few basic gestures.

8.7.2 Gesture recognition

The raw touch API can be used to detect gestures. To illustrate how this works, I created the sample application SimpleGestures. SimpleGestures is shown in figure 8.12 and allows you to change the color of these shapes by performing the specified gesture.

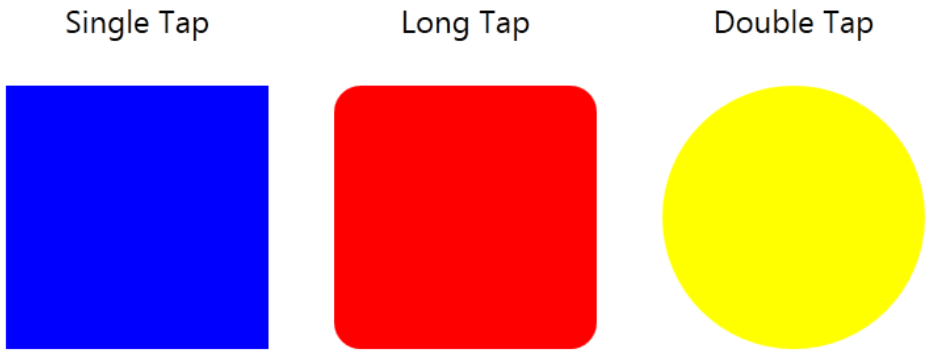


Figure 8.12 SimpleGestures demonstrates how the raw touch API can be used to detect gestures. This application implements the single tap, long tap, and double tap gestures. Perform ing the specified gesture on the shape causes the shape's color to change.

The gestures are implemented as Expression Blend Triggers. This enables them to be reused in different circumstances and provides an easy way to add the tap, long tap, and double tap gestures to any UIElement.

Expression Blend behaviors

Expression Blend behaviors are pieces of reusable code that can be attached to interface objects to enable specific scenarios. The behaviors API is an infrastructure for this code reuse and has three main concepts: Triggers, Actions, and Behaviors.

Triggers are code that wait for a specific condition and then invoke any associated Actions. Actions in turn simply cause something to happen in the application. You could think of Triggers and Actions like cause and effect. They have low coupling and can be mix-and-matched as needed for a particular situation. Behaviors are like triggers and actions rolled into one and are used for situations where high coupling between cause and effect are required.

In this case, we're using Triggers to recognize touch gestures and trigger some other actions or code. Despite the name, Expression Blend behaviors do not depend upon having Expression Blend itself installed and can be used by developers and designers alike. Designers use behaviors to enable interactivity in the Blend design tool, but developers can also make use of the framework in Visual Studio.

You can get started with behaviors by downloading the Expression Blend 4 SDK for .NET 4 from: http://www.microsoft.com/downloads/en/details.aspx?FamilyID=75e13d71-7c53-4382-9592-6c07c6a00207&wa=wsignin1.0&displaylang=en

To add one of these gestures to an interface element, the first thing you need to do is add a reference to System.Windows.Interactivity.dll (from the Blend 4 SDK) and TouchGestureLibrary.dll. Then in your XAML file, add these namespaces:

```
xmlns:i="clr-namespace:
    [CA]System.Windows.Interactivity;assembly=System.Windows.Interactivity"
xmlns:g="clr-namespace:TouchGestureLibrary;assembly=TouchGestureLibrary"
```

Then add the gesture trigger to the interface element you need it on:

```
<Border>
    <i:Interaction.Triggers>
        <g:TapGestureTrigger Tap="TapGesture_Tap" />
    </i:Interaction.Triggers>
</Border>
```

In this case, I opted to make use of the Tap event so I can code the action in my code-behind. In a non-trivial application, you may want to make use of Actions. You would add Actions as a child of TapGestureTrigger.

I do want to show you a few other interesting applications of the raw touch API, but they depend upon the manipulations API as well. We will revisit some applied examples once we have learned about manipulations in chapter 9.

8.8 Summary

In this chapter you have learned about the raw touch API. We saw how the different events, methods, and properties can be used to access information about individual touches. We also created two sample applications that illustrated the API and saw additional applied examples of how the raw touch API can enable more advanced scenarios.

Raw touch is only half of the full WPF Touch API, though. We have not been able to do many advanced controls or really interesting interactions because the raw touch API is too low level. It is great when we

need to respond to each touch, but it is not as good for doing higher level manipulations or interpreting multiple fingers at once. This is where the manipulation API fits in.

For example, recall the Hello, Multi-touch application from chapter 4 where we scaled, panned, and rotated a rectangle around the screen. That was pretty easy to create with the manipulation API. Imagine trying to do the same thing with just the raw touch API. It would be a lot more work, to say the least!

In the next chapter, we will learn the manipulation API and then see how it can really help us easily create very advanced multi-touch interactions. We will also see how you can use parts of the manipulation and raw touch API together to address some advanced scenarios.

Once you have learned the manipulations API, you will have almost all of the tools you need to create any multi-touch WPF application you need. After the manipulations chapter, we will wrap up this part of the book with two chapters on advanced scenarios involving touch visualizations, drag-and-drop, and custom touch devices.

9

Manipulating the interface

In chapter 8, we learned all about the raw touch API in WPF. Raw touch is all about individual touches and is great for certain scenarios, but difficult for others. Most of the applications you create will need to operate on a higher level of abstraction than individual touch events. WPF's manipulation API lets you examine and react to multi-touch manipulations. The manipulation API tracks the aggregate behavior of groups of fingers and reports translation, rotation, and scale transformations caused by the movement of these fingers. The manipulation API also includes inertia, which simulates friction after your flick an object, which is an easy way to add some physicality to your interface.

In this chapter, we are going to learn everything you need to know about the manipulation API. We will start out by learning about some manipulation concepts, how to use the WPF manipulation API, how to create controls using manipulations, and what to do when manipulations reach their limits. We'll also go into the more advanced topics including recognizing gestures with manipulations, coordinating the raw touch API with the manipulation API, and controlling the manipulation on a per touch level.

You already know the raw touch API. Once you learn the manipulation API, you will be well prepared to create almost any type of custom multi-touch interaction in your applications. Let's start by discussing what we mean when we say manipulations.

9.1 Manipulation API overview

The manipulation API includes enhancements to the WPF core classes and adds several supporting classes. It is not a difficult API to learn and once you understand the basics, you will see how the API can be used in a variety of scenarios.

In this section we will cover manipulation API concepts, when to use the manipulation API, and discuss at a high level the API itself. Once we have learned about the manipulation API, we will use the manipulations to create custom multi-touch controls and other advanced scenarios in the rest of this chapter.

Before we get into how to use the API, let's make sure that we understand a few simple concepts behind the manipulation API and what it provides us.

9.1.1 Manipulation API concepts

The WPF manipulation API helps you interpret raw touch input. It tracks raw touches and models them into translation, rotation, and scale transformations. These are the fundamental transformations that can be interpreted unambiguously from arbitrary multi-touch input. Figure 9.1 illustrates these transformations.

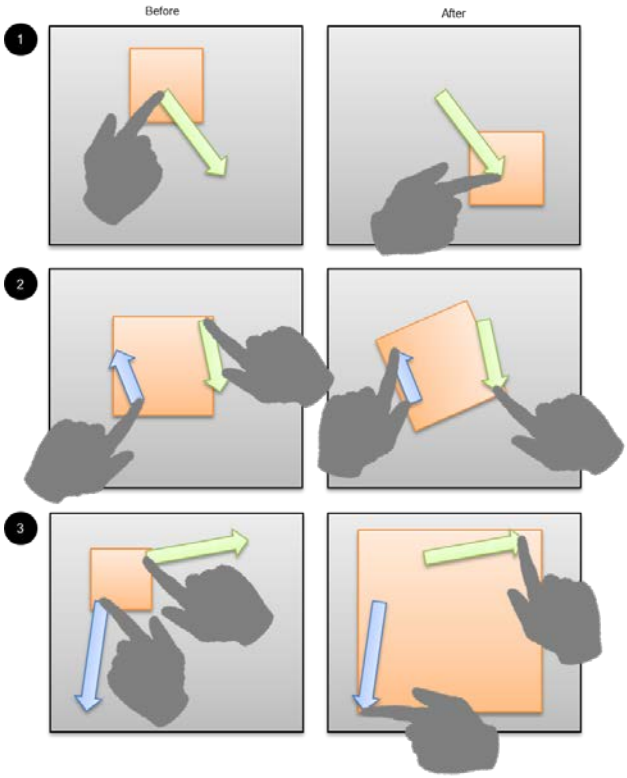


Figure 9.1 The three transformations that the manipulation API provides: translation #1, rotation #2, and scale #3. Rotation and scale normally require at least two fingers, although manipulations can be configured to allow single finger rotation.

Modeling the raw touch information as transformations makes it easier to interpret the input. If you only had access to raw touch input, you would have to interpret the change in x and y position for each finger. This is easy with only one finger, but with multiple fingers it becomes very challenging to interpret the meaning of the large amount of information. With a device that supports 10 touches, you would have to consider up to twenty degrees of freedom for every change in the interface.

The manipulation API eliminates this challenge by reducing the amount of information you need to look at and presenting it in a more useful format. WPF manipulations four degrees of freedom: horizontal and vertical translation, the rotation angle, and the uniform scale value. This is true regardless of the number of fingers participating in the manipulation.

When you use manipulations, you need to figure out how to map these four degrees of freedom of input into degrees of freedom in your interface. It is up to you to decide how you want the interface to respond and the interface response does not necessarily have to match translation, rotation, or scale transformations from the manipulation.

MAPPING INPUT TRANSFORMATIONS TO INTERFACE FEEDBACK

When you process the manipulations input, you need to figure out what will change in the interface in response to the manipulation transformations. Depending upon what you want the user to be able to do, you might map manipulations to several different types of interface values. Figure 9.2 shows how the horizontal translation value of a manipulation can be mapped to three different degrees of freedom of the interface.

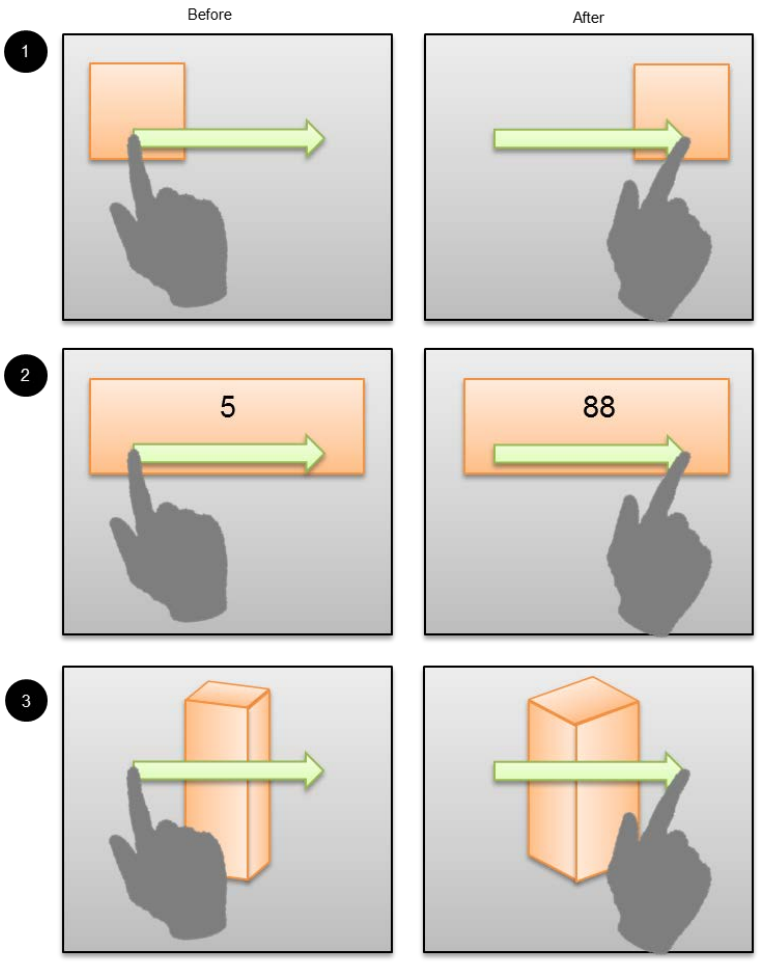


Figure 9.2 A manipulation with horizontal translation can be mapped to several different degrees of freedom in the interface. The same physical movement could cause object translation #1, changes in a data value #2, or rotation of a 3D object #3.

As you can see, you can map the horizontal translation to several different values in your code, each of which provides different but expected feedback to the user. You could potentially map horizontal finger motion to vertical interface movement, but this would not be expected and would just cause confusion.

You can predict how natural a particular mapping is by evaluating how direct the interaction is. Recall from chapter 1 the three components of directness: spatial proximity, temporal proximity, and parallel action. If the proposed interaction keeps the fingers close to the interface visual and moves the visual at the same time and in a similar direction as the finger, then the interaction is likely to be very natural for users.

Manipulations and gestures

In chapter 3, we discussed that the two main ways to interact with a NUI interface are through manipulations and gestures. The key difference between the two is that manipulations are continuous with constant feedback while gestures are discrete with feedback only after the gesture is complete.

The manipulation API is primarily intended to be used for creating manipulations, but it can also be helpful for recognizing certain simple gestures. You can also easily combine manipulations and gestures using the same API.

For example, suppose that you wanted to allow the user to translate an object around the screen but if they flick it to the left it goes into a container on the side of the screen. You would first set up a manipulation to allow the use to translate the object. During the manipulation, you can detect the flick gesture by testing to see if the translation velocity exceeded a threshold in the appropriate direction.

You could also recognize flicks and other basic gestures without allowing manipulation of objects. We will talk more about recognizing gestures with the manipulation API in section 9.5.

The manipulation API is a key component of creating natural interactions. The API models touch input as translation, rotation, and scale transformations. You can map these transformations to any type of interface feedback to create a direct and natural interface.

9.1.2 When to use the manipulation API

The Surface SDK includes controls that package up commonly used behaviors. These are very useful, but in many circumstances you will need to go beyond existing components and build new behaviors and new controls. You will want to use the manipulation API for almost any multi-touch component you create.

In the last chapter, we discussed when to use the raw touch API and when to use the manipulation API. Here are scenarios from that discussion where the manipulation API would be more appropriate than the raw touch API:

- Pan, rotate, or scale a UIElement
- Track continuous linear, rotation, or scale motion
- Recognize a gesture based upon velocity threshold
- Simulate inertia and friction after an interaction

Manipulations are needed for many scenarios and it will be helpful for you to understand how to use them. Manipulations use and consume raw touch events internally, so in most cases you will only use one or the other, but there are some cases where you'll want or need to use both at the same time. In section 9.6 we will discuss how the raw touch and manipulation APIs work together.

Now that we understand at a high level what the manipulation API can be used for, let's learn about the manipulation API itself.

9.2 The manipulation API

The manipulation API consists of two parts: manipulation events, and supporting classes. The manipulation events are available on any UIElement. The supporting classes include several classes that are used to get or set additional information about the behavior of a manipulation as well as the core manipulation and inertia processor classes.

Manipulation events require use of several new EventArg classes. While these classes could technically be considered supporting classes, I will group them with the discussion of the manipulation events. Most of the supporting classes are only used in advanced scenarios while the EventArgs classes are used in all scenarios.

NOTE

The raw touch API is available on UIElement, UIElement3D, and ContentElement. In contrast, the manipulation API is only available on UIElement. You can still manipulate a UIElement3D and ContentElement by setting up the manipulation on a UIElement that contains the UIElement3D or ContentElement.

A manipulation in this API has two phases: the user-controlled manipulation and friction-controlled inertia. In previous versions of the Surface SDK, developers had to deal with separate manipulation processors and inertia processors. With WPF 4, this is no longer the case. Both of these phases are incorporated into a unified manipulation API.

WPF 4 includes on the UIElement class six manipulation events and a property to enable manipulations. There are also new EventArgs classes that support the manipulation events as well as some related classes, enums, and interfaces.

In this section we will learn about the manipulation events, the order in which the events are called, and how you can affect that order. You will spend most of your time with the manipulation API using UIElement manipulation events, so let's cover those first.

9.2.1 Manipulation events

When you use the manipulation API, there is a lot of math and logic that goes on behind the scenes. Fortunately, we only need to worry about a small set of events available on any UIElement.

You can enable manipulations on as many UIElements as you want. Each UIElement can only have one manipulation occurring at a time, though. You can have a manipulation on one UIElement and a different manipulation on a child UIElement at the same time. Multiple independent manipulations on different UIElements can occur simultaneously through the magic of multi-touch!

Table 9.1 summarizes the manipulation events that are available on UIElements.

Table 9.1 The Manipulation API provides these events on all UIElements

Event	Description
ManipulationStarting	Occurs when the manipulation is starting but before any calculations occur
ManipulationStarted	Occurs after the manipulation has been initialized
ManipulationDelta	Occurs when any of the captured manipulators move during the manipulation and during inertia
ManipulationInertiaStarting	Occurs when the last manipulator has removed, ending the manipulation phase and starting the inertia phase
ManipulationCompleted	Occurs when both of the manipulation and inertia are complete
ManipulationBoundaryFeedback	Occurs when a manipulation encounters a boundary

The ManipulationDelta event is probably the most important event for you to subscribe to. It is raised every time the manipulation changes and gives access to the current state and latest changes (or delta) in each of the manipulation transformations: translate, rotate, and scale.

The manipulation events each have a specific purpose and are raised in a specific order. Depending upon what you need to do with the interface, you can affect the event sequence by skipping some events or repeating some. Let's discuss the sequence in which manipulation events are called.

9.2.2 Manipulation event sequence

Manipulation events occur in a specific order. Figure 9.3 shows the sequence of events for a typical manipulation. There are also some variations on this path, which we will discuss shortly.

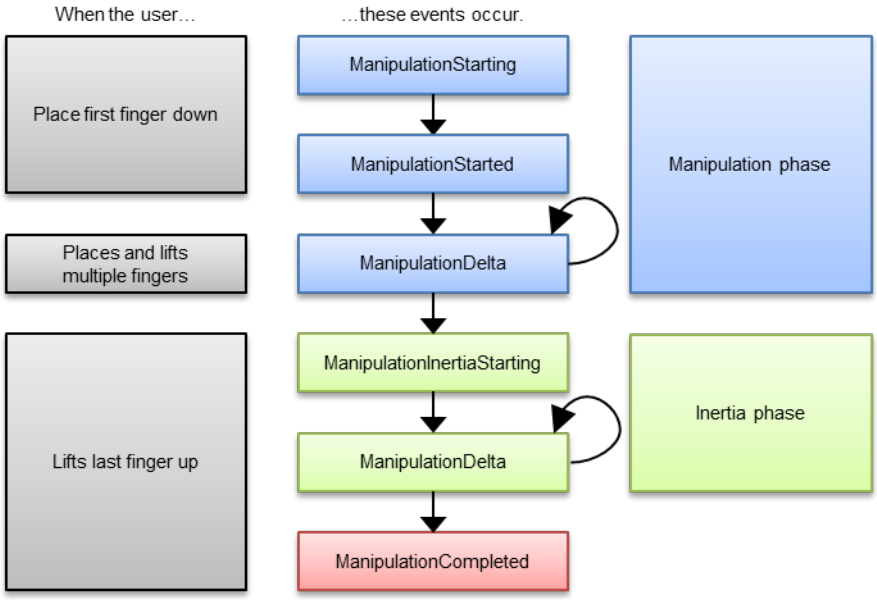


Figure 9.3 The standard sequence of manipulation events is controlled by the user's fingers. The events are categorized into two phases. The manipulation phase occurs while fingers, or manipulators, are down and is followed by the inertia phase when the last manipulator has been lifted.

Manipulations in WPF are split into two phases: manipulation and inertia. The manipulation phase starts when the first manipulator is captured and ends when all the manipulators have released or been removed programmatically. The inertia phase begins when the manipulation phase ends.

Inertia adds some physical behavior to the interaction. During a typical inertia phase the translation, rotation, and scale transformations continue with the same velocity they had when the manipulation phase ended. You can also specify your own initial velocities. Inertia simulates friction by subjecting these transformations to deceleration. When the inertia movement slows to zero, the inertia phase ends and the entire manipulation sequence is complete.

NOTE

You may notice that the ManipulationBoundaryFeedback event is not included in the event sequence. This event is only raised when you report feedback in a ManipulationDelta event using the ManipulationDeltaEventArgs.ReportBoundaryFeedback() method. We will be discussing boundary feedback in section 9.4.

Notice that the same ManipulationDelta event is called during both the manipulation phase and the inertia phase. This allows you to easily reuse the same code since you'll likely want to use the same mappings of manipulation data to interface feedback during both phases. You can tell whether you are in the inertia phase by checking the ManipulationDeltaEventArgs.IsInertial property.

Most of the work is done in the ManipulationDelta event, but the other events are also important. The other events are used to configure the behavior and context of the manipulation to ensure that your ManipulationDelta event handler will get good data.

CONFIGURING MANIPULATION BEHAVIOR

The behavior of a manipulation is generally configured and controlled by setting properties or calling methods on the EventArgs of a particular manipulation event. For example, during the ManipulationStarting event, you may need to set the ManipulationStartingEventArgs.ManipulationContainer property to a non-moving visual parent. (I'll explain why you need to do this in section 9.3.4.)

Similarly, to configure the parameters that control inertia, you'll need to set some ManipulationInertiaStartingEventArgs properties during the ManipulationInertiaStarting event.

Depending upon what properties you set or what methods you call during various the manipulation events, you might skip or repeat some events in manipulation event sequence. As we go through each of the manipulation events, I'll point out places where the event sequence can be changed.

CANCELLING AND COMPLETING MANIPULATIONS

In most cases when you set up manipulations, you'll let the event sequence play out to completion. There are some cases where you will want to cancel the entire manipulation or skip to the end of the manipulation. You can trigger these actions by calling e.Cancel() or e.Complete() in a manipulation event. As an example of when this would be useful, imagine you are manipulating an object and elsewhere in the interface you change modes so that object shouldn't be manipulated any more. In that case you might want to complete the manipulation early.

All manipulation events provide the Cancel() method in the EventArgs, but the Complete() method is only in ManipulationStarted and ManipulationDelta. Both of these methods immediately end the manipulation and skip inertia, but there is a subtle difference between the Cancel() and Complete(), though, illustrated in figure 9.4.

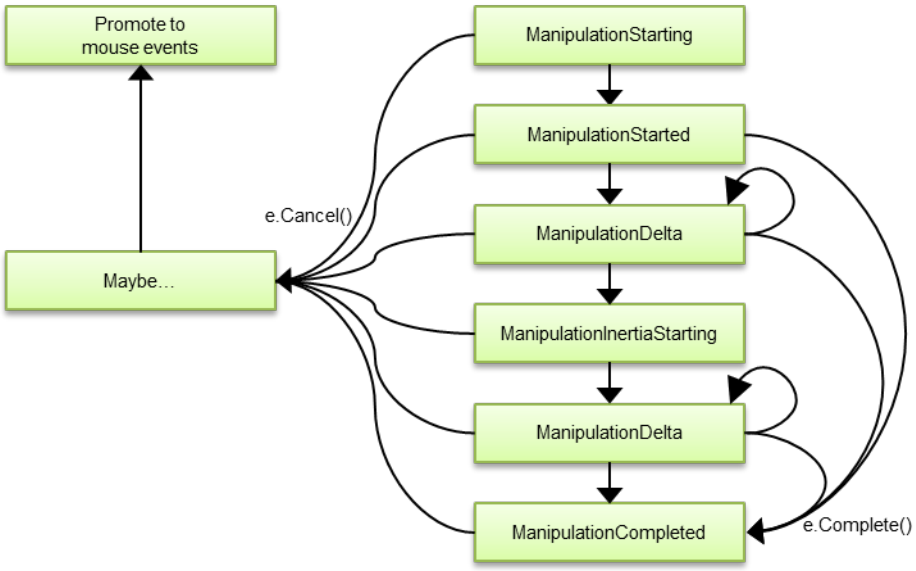


Figure 9.4 Most events allow you to manually complete the manipulation. All events allow you to cancel the manipulation. Completing a manipulation skips or ends inertia and jumps to the ManipulationCompleted event. Cancelling a manipulation ends the manipulation without inertia or any further manipulation events.

Calling Complete() basically says "this manipulation is good, but it is time to end it" and will skip inertia and raise the ManipulationCompleted event. This can be used when an element reaches a boundary to instantly stop it.

In contrast, calling Cancel() says "I wish this manipulation was never started in the first place!" and in some cases will promote any processed touch events to mouse events instead. The mouse events will replicate the movement and status of the primary touch point from the start of the manipulation until it was cancelled.

Figure 9.4 shows "maybe" during the Cancel() branch because the rules for whether a cancelled manipulation is promoted to mouse events are somewhat complicated. (We discussed mouse promotion in chapter 8.) Mouse promotion from a manipulation will only occur if all of the following conditions are true:

- Inertia has not started yet
- The manipulation includes the primary touch point TouchDevice

- The TouchDevice supports mouse promotion

As with regular mouse promotion, only the primary touch point can be promoted to mouse. The default StylusTouchDevice supports mouse promotion, but custom TouchDevices may not support mouse promotion.

Cancelling a manipulation would only be necessary if you are creating an application that mixes touch-aware and mouse-only controls. This might be the case when upgrading a legacy application to include multi-touch features. The best user experience is achieved when designing the interface for touch from the beginning, so I don't anticipate many people needing to cancel manipulations and depend upon mouse promotion.

The Cancel() method is available during the entire manipulation sequence, but the Complete() method is only available with ManipulationStarted and ManipulationDelta. It makes sense that you can't complete the manipulation in the ManipulationStarting event, because at that point it hasn't started. In addition, during the ManipulationCompleted event the manipulation has already completed.

So why is ManipulationInertiaStarting left out? Actually, it is not. As we will see, if you do not change the default parameters in ManipulationInertiaStarting, then the next event will be ManipulationCompleted. If you want to skip inertia and complete the manipulation from ManipulationInertiaStarting, simply return without doing anything or reset the inertia parameters to their default values.

Now that we understand the order of events during a manipulation, we're ready to learn about how to use each of these events. Each event has an important role in allowing you to configure the manipulation behavior or giving you information about the manipulations.

9.3 Manipulation recipes

At this point, you should understand at a high level how manipulations work and when each event is called. We still need to learn how to use them in real scenarios, though. I want to focus on practical scenarios, so I decided to organize this section into "how to" scenarios. You could think of these scenarios as recipes. We will mix together different ingredients (features of the manipulations API) and achieve some interesting results.

In this section, we're going to focus on recipes that only use the basic ingredients: manipulation events. In later sections we will look at advanced scenarios using other parts of the manipulations API. We can achieve a lot with just the events and in many cases, these are all you need.

Let's get cooking, err, coding!

9.3.1 Enabling manipulation events

Manipulations are disabled by default. If you want to enable manipulations on a particular UIElement, you need to do is set the UIElement.IsManipulationEnabled property to true. This will cause that element to automatically capture TouchDevices and process their movement into manipulation events.

> **NOTE**
>
> Internally, the WPF manipulation processor requires input to implement the IManipulator interface. WPF touches are provided by a TouchDevice, which implements IManipulator. In advanced scenarios, you can also include custom classes that implement IManipulator in manipulations. For this reason, touches or other inputs that participate in a manipulation are generally referred to as manipulators.

Of course, just processing the manipulations internally doesn't do much. We also want to be notified when manipulation events occur. To do this, you'll need to subscribe to the manipulation events.

SUBSCRIBING TO MANIPULATION EVENTS

You can subscribe to these manipulation events on any UIElement or derived type. As long as IsManipulationEnabled is true, your event handlers will be called at the appropriate time in the manipulation sequence.

Just like any other WPF control, you can use either XAML or code to subscribe to each event. For example, to subscribe to the ManipulationDelta event, you could use this snippet in XAML:

```
<Border Name="border" ManipulationDelta="border_ManipulationDelta">
```

or this snippet in code:

```
border.ManipulationDelta += border_ManipulationDelta;
```

Both approaches accomplish the same thing and which approach you use depends upon the context of the situation. I prefer to use XAML when I can, but you might need to use code in custom controls or dynamic situations.

You can configure multiple UIElements for manipulation. This means that you may have a manipulable UIElement as the child of another manipulable UIElement. In those cases, it is important to handle the events appropriately.

HANDLING THE MANIPULATION EVENTS

The manipulation events are all routed events and use the bubbling routing strategy. This means that they will first be raised on the UIElement that is being manipulated. If the events are not handled they will be raised on the parent element and then the next parent element (bubble up) until the events are handled or reach the root element.

You don't need to subscribe to manipulations for every single UIElement. You only need to subscribe to and handle events at a level that matches what is being manipulated. Figure 9.5 illustrates a custom control that moves itself within a parent Canvas by handling the ManipulationDelta event.

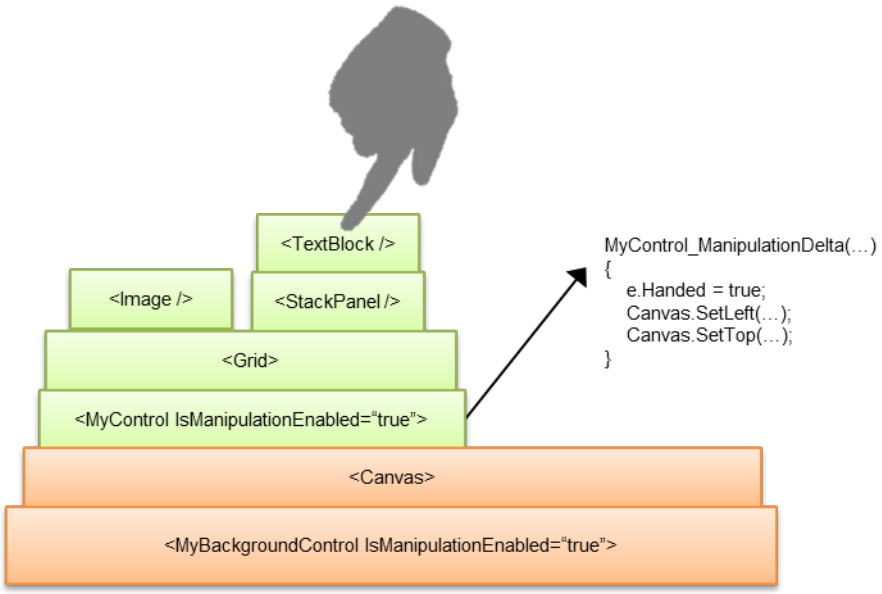


Figure 9.5 MyControl, a custom control, has several visual children and IsManipulationEnabled set to true. MyControl also has a ManipulationDelta event handler that updates the Canvas Left and Top properties so that it will move within a parent canvas. The ManipulationDelta event is called on MyControl even when the user is directly touching a child visual element. Notice that because MyControl handles ManipulationDelta, the Canvas and MyBackgroundControl do not receive this event.

You'll want to handle manipulation events in almost every case to prevent a parent manipulation from accidentally processing the wrong information. In the example in figure 9.5, the Canvas is within another custom control called MyBackgroundControl that is also manipulation enabled and has a unique manipulation behavior. If MyControl did not handle the ManipulationDelta event, then MyBackgroundControl would also receive the event resulting in both controls reacting to the same manipulation, probably in a non-intuitive way.

In this scenario, MyControl configures itself for manipulations and is handling the manipulation events, but there is no reason why the control being manipulated has to be the one that sets up the manipulations and handles the events. MyControl could have enabled manipulations and subscribed to manipulation

events on one the controls in its control template such as the Grid. The Grid itself is not aware that it is being manipulated because MyControl handles everything.

Similarly, it would be valid for an ItemsControl to set IsManipulationEnabled to true for all visual children and subscribe to each child's manipulation events. This would let the ItemsControl control the manipulation behavior of each child relative to the others. In fact, this is how the ScatterView control works internally.

> **ScatterViewItem manipulation events**
>
> In chapter 6 we learned that ScatterViewItems are controlled by manipulations. They expose three events that let you hook into the manipulation process: ContainerManipulationStarted, ContainerManipulationDelta, and ContainerManipulationCompleted. These events have the same meaning as their non-container equivalents.
>
> Since the SVI handles updating its position from the manipulation internally, you are not required to do anything in these events. They are useful, though, when integrating the behavior of ScatterViewItems with the rest of the application. This occurs most often with drag-and-drop operations, which will be covered in chapter 10.

Now that we know how to enable and subscribe to manipulation events, we need to do something with the events. Let's start with something very simple: changing a number using a manipulation.

9.3.2 Change a number using manipulations

If you think about it, most interface interactions involve the user providing some input and the computer changing a number. The number usually represents something, such as an aspect of the interface or some user content. For example, moving around a ScatterViewItem involves manipulating the numbers that control its position, while manipulating a slider might change a linked number that represents the price of a product. Manipulations fall into this pattern and any time you create a manipulation, you will end up changing a number.

Let's embrace that and start with a very simple example of mapping the manipulation to a number in the interface. Even though it seems simple, this capability is very powerful. In later examples we will map the manipulation to more practical feedback. Figure 9.6 shows what the interface of the NumbersManipulation sample will look like.

Figure 9.6 The NumbersManipulation sample lets you change the number by touching a green rectangle and moving fingers. Moving right increases the number while moving left decreases the number.

The best thing about this example is that it only needs one manipulation event: ManipulationDelta. Some manipulation events exist to get information from you, and some exist to give you information. The

ManipulationDelta event is the kind that gives you information. It tells you almost everything you need to know for any responding to manipulation input. Let's see how to use it.

CREATE THE VISUALS

In order to observe the changing number, we will have to create some visuals to display the it. Create a new WPF Surface application called NumberManipulation and add this to SurfaceWindow1.xaml inside the Grid element:

```
<StackPanel HorizontalAlignment="Center">
    <TextBlock Name="txtNumber"
               Text="100"
               FontSize="48" />
    <Border Name="border"
            Background="Green"
            Width="600"
            Height="200" />
</StackPanel>
```

We need to store a number that we can update later, so in SurfaceWindow1.xaml.cs add this property:

```
double _number = 100.0;
double Number
{
    get { return _number; }
    set
    {
        _number = value;
        txtNumber.Text = _number.ToString("F1");
    }
}
```

Now whenever we set the Number property, the TextBlock will be updated accordingly.

SUBSCRIBE TO MANIPULATIONDELTA

The point of this exercise is to manipulate the number, so let's set up a manipulation. In SurfaceWindow1.xaml, we already added the Border element that we will use for manipulation. All we need to do now is enable the manipulation and subscribe to the ManipulationDelta event. Update the Border element to look like this:

```
<Border Name="border"
        Background="Green"
        Width="600"
        Height="200"
        IsManipulationEnabled="True"
        ManipulationDelta="Border_ManipulationDelta" />
```

We could have done this in the code-behind. Adding this code in the constructor after the call to InitializeComponent() would do the same thing.

```
border.IsManipulationEnabled = true;
border.ManipulationDelta += border_ManipulationDelta;
```

Notice that I made sure to set IsManipulationEnabled to true. If you don't, your manipulation events will never be called.

ADD THE EVENT HANDLER

Next, you need to add an event handler for the ManipulationDelta event. If you are following along, this event may have already been created if you took advantage of the Intellisense auto-complete. The event handler should look like this:

```
void border_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{ }
```

This event handler will be called every time the user moves any finger that is captured to this manipulation. It will also be called repeatedly during the inertia phase until the inertia has completed.

RESPOND TO THE CHANGE IN MANIPULATION

Finally, you need to add some code to the ManipulationDelta event handler to respond to changes in the manipulation. You do this by using the properties of the provided ManipulationDeltaEventArgs instance.

ManipulationDeltaEventArgs

The ManipulationDeltaEventArgs class holds all of the data about what has changed since the last ManipulationDelta event as well as since the manipulation started. It also gives you some information about the context of the manipulation. Table 9.2 shows all of the properties of this class.

Table 9.2 should be in sidebar

Table 9.2 ManipulationDeltaEventArgs exposes the following properties

Name	Type	Description
CumulativeManipulation	ManipulationDelta	Gets the cumulative changes since the start of the manipulation
DeltaManipulation	ManipulationDelta	Gets the changes since the last ManipulationDelta event
IsInertial	bool	Gets whether event occurs during the inertia phase
ManipulationContainer	IInputElement	Gets the container that defines the coordinates for measuring the manipulation
ManipulationOrigin	Point	Gets the center of the manipulation. This is the average of all manipulator positions.
Manipulators	IEnumerable<IManipulator>	Gets the collection of manipulators. Typically the manipulators are derived from TouchDevice.
Velocities	ManipulationVelocities	Gets the most recent rate of change of the manipulation

Most of the time the only thing you will need out of the ManipulationDeltaEventArgs is the DeltaManipulation property. This property is of type ManipulationDelta (which we will look at in a second) and represents the changes in the manipulation since the last ManipulationDelta event. This means that this data is appropriate for making the interface change during the manipulation. This is a key part of creating direct manipulation interactions.

In this case, all we need to do is add this line to the event handler:

```
Number += e.DeltaManipulation.Translation.X;
```

This will increase the number if the manipulation is translating to the right or decrease the number if the manipulation is translating to the left. The manipulation occurs if you touch the green Border element. You can run the application and test it now.

TEST OTHER MANIPULATION TYPES

There are a couple different ways we could have chosen to manipulate the number. As we discussed in section 9.1.1, in order to use manipulations you need to map a degree of freedom of the manipulation to a degree of freedom in the interface. In this case, horizontal movement maps to changing the number. We could easily change this to use vertical movement by adding `e.DeltaManipulation.Translation.Y` to the number instead, or we could make the number change based upon the change in rotation or scale. These values are all part of the ManipulationDeltaEventArgs.DeltaManipulation property, which is an instance of the ManipulationDelta class.

The ManipulationDelta class

The purpose of ManipulationDelta class is to hold changes in translation, rotation, and size manipulation values. Table 9.3 shows the properties of the ManipulationDelta class.

Table 9.3 should be in sidebar

Table 9.3 The ManipulationDelta class exposes these properties

Property	Type	Description
Translation	Vector	Change in horizontal and vertical movement
Rotation	double	Change in rotation around the manipulation origin
Scale	Vector	Change in scale expressed as a multiplier. A value of 1 is no change.
Expansion	Vector	Change in scale expressed in device-independent pixels. A value of 0 is no change.

These are all the average changes. For example, if the manipulation is tracking two fingers, the may have translated horizontally by different values but only a single horizontal translation is reported. Similarly, rotation and scale only report a single set of values whether the number of fingers involved are two or ten.

Scale and expansion both measure changes in size by two or more fingers. You should use scale when you need to update a ScaleTransform or multiply a length and use expansion when you need to change the Width or Height of a visual element. Although the scale and expansion properties are Vectors, the X and Y values are always the same. This is because in WPF, manipulations always use uniform scaling.

The ManipulationDelta class is primarily used in the ManipulationDeltaEventArgs class for the DeltaManipulation and CumulativeManipulation properties.

The `ManipulationDeltaEventArgs.DeltaManipulation` property contains the changes since the last call to the ManipulationDelta event. You will want to use this property for changing values that should continuously and directly track the manipulation.

The `ManipulationDeltaEventArgs.CumulativeManipulation` property contains the changes since the manipulation started. It is useful for detecting thresholds such as whether the manipulation has stayed within 20 pixels of the starting position. You won't need this property as much as the DeltaManipulation property, but it is really useful in certain scenarios. We will talk use this in a later example in this chapter.

Let's try a few other manipulation mappings. Change the ManipulationDelta event handler to have this line instead:

```
Number += e.DeltaManipulation.Rotation;
```

Now when you run the program, you'll need to use two fingers and rotate them to change the number. The Rotation value is returned in degrees. Take note of this because if you even need to use this value in any trigonometric functions such as Math.Sin() or Math.Cos(), you'll have to convert it to radians:

```
double rotationRadians = e.DeltaManipulation.Rotation * Math.PI / 180.0;
```

Finally let's test the behavior of scale. Table 9.3 described how the ManipulationDelta class has the scale and expansion properties. These both measure the same thing but expose it in two different ways. Accordingly, you need to use them in slightly different ways. You should use Expansion by adding the value, similar to Translation and Rotation:

```
Number += e.DeltaManipulation.Expansion.X;
```

If there is no change, then the Expansion value is zero. With scale, no change is indicated by a value of one. This is because Scale is a multiplier and you need to multiply your target value by the Scale value like this:

```
Number *= e.DeltaManipulation.Scale.X;
```

One way to think of the difference between Expansion and Scale is this: Expansion says "the manipulation is this many device-independent units larger (or smaller) since the last delta event" while the Scale says "the manipulation is this percent larger (or smaller) than the last delta event." Most scenarios end up using Scale because it is flexible enough to map to data such as matrix transformations that are not measured in device-independent units.

Now that we have covered the basic scenario of manipulating a number, let's move on to other more useful scenarios (or recipes) where the manipulation is mapped to more interesting values. We will also see how to use the other manipulation events.

9.3.3 Manipulate the position of a visual

It is very common to map a manipulation to various visual parameters such as position, size, and orientation. For now, let's focus on just the position.

There are two primary ways that you can update the position of a visual. First, you could wrap it in a Canvas and change the Canvas.Left and Canvas.Top values. This works well but depends upon the visual being contained in a Canvas. If you are designing a control, you may not be able to guarantee that. This approach also invalidates part of the visual layout and needs an expensive layout pass.

The second way to change the position of a visual is updating a TranslateTransform within a RenderTransform. This is reliable and relatively high performance. It also does not depend upon any other visual hierarchies.

RenderTransform review

The RenderTransform is a property on UIElement that can be set to one or more types of transformations including TranslateTransform, RotateTransform, and ScaleTransform, plus TransformGroup that can contain other Transforms. RenderTransform affects the render position of the UIElement. It is applied after the layout pass is complete so it does not affect the layout or render sizes.

Here is an example of using a TranslateTransform in XAML to change the rendered position of an element:

```
<Border>
    <Border.RenderTransform>
        <TranslateTransform X="15"
                            Y="-45" />
    </Border.RenderTransform>
 [Other content]
</Border>
```

Code should be in sidebar

In comparison, LayoutTransform will transform visuals during the layout pass and can affect the positions of other visual elements. This is slower than using a RenderTransform, which only transforms elements after the layout is determined. RenderTransform is usually recommended for our purposes.

Let's see how to change a position using a TranslateTransform inside of a RenderTransform.

REFERENCING THE TRANSLATETRANSFORM

You will need to get a reference to the TranslateTransform you want to change. If you have defined the transform in XAML, you could give it a name (using x:Name syntax):

```
<Border.RenderTransform>
    <TranslateTransform x:Name="translate"
                        X="15"
                        Y="-45" />
</Border.RenderTransform>
```

If you already have a reference to the visual element that is being transformed, you can also set the RenderTransform property to a new TranslateTransform as an initialization step and store a reference for later:

```
this.translate = new TranslateTransform();
border.RenderTransform = translate;
```

Be careful when setting the RenderTransform. If there is already a transform setup by XAML or other code, you may overwrite it. You should only change the RenderTransforms for UIElements that your code created and you control completely.

Once you have the reference to TranslateTransform, you can get or set the X and Y properties. For this scenario, we are going to want to update these using the manipulation.

UPDATING THE POSITION

In section 9.3.1, we updated the value of a property in the ManipulationDelta event handler. Here we will do almost the same thing, except we'll update the TranslateTransform instead.

```
void Border_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    TranslateTransform translate = GetTranslateTransform();
    translate.X += e.DeltaManipulation.Translation.X;
    translate.Y += e.DeltaManipulation.Translation.Y;
    e.Handled = true;
}
```

Now, as long as the transform was set to the visual's RenderTransform, you will set movement. If you are transforming the same visual that you are manipulating, then you will get jumpy flickering movement. This is because ManipulationDelta reports values relative to a frame of reference. By default, the frame of reference is the immediate visual element, and since you are moving the frame of reference under the fingers, the ManipulationDelta gets faulty data.

The solution is to set the frame of reference to a non-moving visual. We can do this by setting the ManipulationContainer in the ManipulationStarting event, which will be discussed in the next section.

9.3.4 Customize manipulation behavior

All manipulations operate in the same basic manner but you can customize certain aspects. The customization includes which transformations are enabled, what to use as the frame of reference for measurement, and the rotation point (or pivot) when the manipulation only has a single finger.

You can configure these custom behaviors by setting properties in the ManipulationStarting event, which is called once at the beginning of a manipulation. Once the properties are set and the ManipulationStarting event handler returns, the settings are fixed for that manipulation. After the current manipulation is complete, future manipulations will call ManipulationStarting again and can be customized again with the same or different settings. The ManipulationStarting event uses ManipulationStartingEventArgs, shown in figure 9.7.

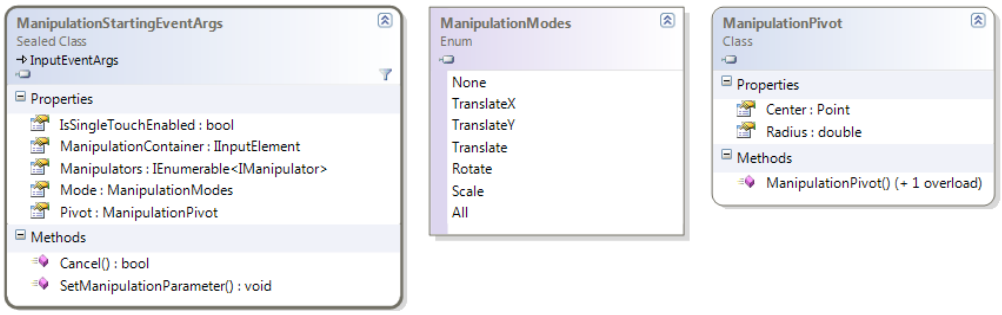


Figure 9.7 The ManipulationStartingEventArgs class lets you set several properties to configure the behavior of the manipulation. Included in this diagram is the ManipulationModes enumeration, which enumerates different transformation modes, and the ManipulationPivot class, which lets you configure single touch rotation.

To use the ManipulationStarting event, all you need to do is set properties of ManipulationStartingEventArgs that do not have good default values. These properties are shown in table 9.4.

Table 9.4 ManipulationStartingEventArgs exposes these properties

Property	Type	Default	Description
IsSingleTouchEnabled	bool	true	Determines whether a single finger can initiate a manipulation. If set to false, manipulations require two fingers before they will start.
ManipulationContainer	IInputElement	Same as sender	ManipulationContainer defines the point of reference for measuring changes in fingers. Defaults to the manipulated element.
Manipulators	IEnumerable<IManipulator>	-	Collection of manipulators participating in this manipulation
Mode	ManipulationModes	All	Determines which transformations are allowed during this manipulation
Pivot	ManipulationPivot	Null	Configures the behavior of single finger rotation

Most of these properties are pretty easy to understand. In most cases, you'll leave IsSingleTouchEnabled to the default of true. Setting it to false will just delay the manipulation from starting until there are two or more manipulators.

The Manipulators property simply contains a collection of the manipulators that are a part of the manipulation. Usually this will be a single manipulator in the ManipulationStarting event but in other events if may contain many manipulators depending upon how many fingers the user is using on the manipulation element.

ManipulationStarting or ManipulationStarted

The ManipulationStarted event seems very similar to the ManipulationStarting event, so their roles might be confusing. You'll most often use ManipulationStarting to configure the manipulation as well as detect when a manipulation starts. ManipulationStarted occurs after the initial manipulation values are calculated and just before the ManipulationDelta is called.

There are really not many, if any, scenarios where you will need to use the ManipulationStarted event. All the information in that event is available in either the ManipulationStarting event or the first ManipulationDelta event.

To access the ManipulationStarting event, all you need to do is subscribe to it on one of you UIElements, typically the same element that has IsManipulationEnabled set to true. For example, in XAML you could write:

```
<Border Name="Border"
        IsManipulationEnabled="True"
        ManipulationStarting="Border_ManipulationStarting"
        ManipulationDelta="Border_ManipulationDelta">
```

Or you could do the same in code:

```
Border.ManipulationStating += Border_ManipulationStarting;
```

If Intellisense didn't already add it, you will also need to add an event handler ManipulationStarting event handler with this signature:

```
void Border_ManipulationStarting(object sender,
                                 ManipulationStartingEventArgs e)
{ }
```

Inside the event handler you can customize the manipulation by setting properties on the provided ManipulationStartingEventArgs instance. Let's discuss how to configure the transformation modes, the manipulation container, and single finger pivoting.

CONFIGURING TRANSFORMATION MODES

This one is pretty easy. ManipulationStartingEventArgs.Mode is a bitmask that represents which transformations the current manipulation will track. It defaults to ManipulationModes.ALL. Table 9.5 lists all of the modes in the ManipulationModes enumeration.

Table 9.5 The ManipulationModes enumeration is used to enable transformations

ManipulationMode	Meaning
All	(Default) Includes Rotate, Scale, and Translate
None	No transformations
Rotate	One or more finger rotation
Scale	Two or more finger scaling
Translate	One or more finger translation in X and Y
TranslateX	Translation in X
TranslateY	Translation in Y

The ManipulationModes can be combined as necessary. For example, if you only wanted to enable Rotate and Scale but not Translation, you could add this line to the ManipulationStarting event handler:

```
e.Mode = ManipulationModes.Rotate | ManipulationModes.Scale;
```

Similarly, you could enable just TranslateX or TranslateY, or both by using Translate. These two lines of code are equivalent:

```
e.Mode = ManipulationModes.TranslateX | ManipulationModes.TranslateY;
e.Mode = ManipulationModes.Translate;
```

In the ManipulationDelta event handler, the disabled modes will have delta measurements of zero. For the enabled modes, the delta measurements will be relative to the ManipulationContainer.

CONFIGURING THE MANIPULATIONCONTAINER

The ManipulationContainer is used as a frame of reference for the manipulation measurements. It is important to set the ManipulationContainer to a good frame of reference for two reasons:

1. Choosing a convenient frame of reference will make it easier to use the data that you get in ManipulationDelta. For example, if you are translating a Border within a Canvas, setting the ManipulationContainer to the Canvas will result in values relative to the Canvas. These values can be used immediately in Canvas.SetLeft and Canvas.SetTop without further transformation.
2. If the manipulation changes the position of the ManipulationContainer then your manipulations will result in flickering. We saw this phenomenon in chapter 4 before we set the ManipulationContainer to the parent. The flickering is caused because the motion of the manipulators is being measured relative to the ManipulationContainer, which itself is moved based upon these measurements. This forms an unstable feedback loop resulting in the object flickering back and forth.

The ManipulationContainer for a manipulation defaults to the element with IsManipulationEnabled set to true. If the manipulation updates the position of this element then you will want to set the ManipulationContainer to a frame of reference that is not being moved by that manipulation, such as the parent element.

We configure the ManipulationContainer in the ManipulationStarting event. This is actually the most common thing that you will do in the ManipulationStarting event. Here is an example of how to set the ManipulationContainer of a Border to a parent element named parentPanel:

```
void Border_ManipulationStarting(object sender,
                                 ManipulationStartingEventArgs e)
{
```

```
    e.ManipulationContainer = parentPanel;
    e.Handled = true;
}
```

This code forces the manipulation for the border to be measured relative to the parentPanel. For example, if the border is translating and rotating all over the screen, the deltas reported in ManipulationDelta will be relative to the non-moving parentPanel. As an alternative to hard coding the parent element, you could also cast the sender to a FrameworkElement and set it to the element's parent:

```
var element = sender as FrameworkElement;
e.ManipulationContainer = element.Parent as IInputElement;
```

This is more general and should work in most cases.

One scenario where it is important to set the Mode and the ManipulationContainer is when you want to configure single finger rotation.

CONFIGURING SINGLE FINGER ROTATION

By default, manipulations with a single finger are enabled but you cannot perform a rotation with a single finger unless you configure the manipulation to do so in the ManipulationStarting event handler. Single finger rotation is useful if you want to give the manipulation a more realistic physical behavior or if you want to rotate fixed items without translation.

We need to set the Pivot property to configure single finger rotation. This property is all about measuring transformations, just like ManipulationContainer. In this case, though, we're just measuring rotation.

Typically the manipulation rotation is measured as the rotation of manipulators around the ManipulationOrigin, which is the average position of all manipulators. When you only have one manipulator, then its position is the same as ManipulationOrigin and no rotation can be measured. What if you want to allow single finger rotation?

This is where the Pivot property is needed. The Pivot specifies the center of rotation when there is only a single manipulator or when the manipulation is pinned. (I'll talk more about pinned in a second.)

To help you understand the Pivot property, I created a sample application called PivotManipulations. This application contains a single rectangular Border and it allows you to spin it around the center with a single finger. Figure 9.8 shows this simple interface.

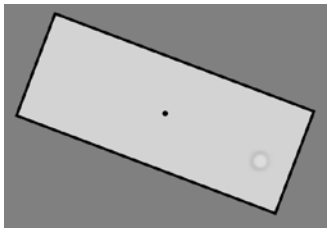


Figure 9.8 The PivotManipulations sample application lets you spin a rectangle around the center pivot, represented the small black dot. A touch visualization is also shown.

In the PivotManipulations application, the name of the Border is "border" and the top-most Grid is named "LayoutRoot". The Border has manipulations enabled and subscribes to the ManipulationStarting and ManipulationDelta events. Listing 9.1 shows the ManipulationStarting event handler.

Listing 9.1 The ManipulationStarting event sets ManipulationContainer and Pivot

```
void border_ManipulationStarting(object sender,
                                 ManipulationStartingEventArgs e)
{
    e.ManipulationContainer = LayoutRoot;

    Point center = new Point(border.ActualWidth / 2,
                             border.ActualHeight / 2);
    center = border.TranslatePoint(center, LayoutRoot);      #A
    double radius = 0;
    e.Pivot = new ManipulationPivot(center, radius);
```

```
        e.Mode = ManipulationModes.Rotate;

        e.Handled = true;
    }
    #A Make center relative to LayoutRoot
```

This code calculates the center of the border and then translates it to be relative to LayoutRoot, which is the ManipulationContainer. The translated point is used to create a ManipulationPivot object and set the Pivot property. Table 9.6 explains the properties of the ManipulationPivot class.

Table 9.6 ManipulationPivot exposes these properties

Property	Type	Description
Center	Point	Determines the center of rotation for measuring manipulators
Radius	Double	Determines a dead around around the center where the effect of rotation tapers off as the manipulator gets closer to the center.

When you run this you can touch anywhere within the border and start spinning it around its center. As you move your finger, the rotation angle will be measured against the configured pivot position. The rotation is also relative to the pivot point when using multiple fingers, but only because this manipulation is pinned.

A manipulation is pinned when you have set the Pivot point and have enabled rotation but not translation. (This is the special case for setting Mode I mentioned a few pages ago.) It is like someone stuck a pushpin through the pivot point. The object can spin freely around the pin and can even scale, but it cannot translate.

The concept of pinned is important because when the manipulation is not pinned, the pivot point will only work when there is only one manipulator. To try this out, put two fingers on the rectangle and rotate them around each other but not the pivot point. The rectangle doesn't spin, although it might wiggle a little bit. This is expected behavior for a pinned object.

To experiment without pinned mode, try commenting out the line that sets e.Mode. Now translate will be enabled for the manipulation. We haven't changed the ManipulationDelta so the rectangle won't actually translate. It will just ignore that information. Single finger rotation still works, but two fingers can rotate the rectangle by rotating around the ManipulationOrigin rather than the pivot point.

Another thing to experiment with is setting the radius. With the radius set to zero, the manipulation is sensitive to rotation all the way up to the center. Touching near the center may produce large unintentional change in the rotation value, which is typically undesirable. You can get around this by setting radius to a larger value.

We've now learned about the ManipulationStarting event and also went into some details about how to customize the ManipulationContainer and Pivot properties.

In section 9.3.3 we learned how to translate an element with manipulation and in this section we learned how to configure the manipulation so it behaves well. Let's move on to another common manipulation scenario: inertia.

9.3.5 Adding inertia

Newton's first law of motion states that an object at rest will stay at rest and an object in motion will not change its velocity except when acted upon by an unbalanced force. We do not live in frictionless, zero-gravity environments, though. Our everyday experiences with manipulating real objects involve inertia but also friction. We can easily add inertia and friction to WPF manipulations.

If you do not configure inertia then manipulations will stop immediately when the last manipulator has been lost, regardless of the velocity of the manipulation. When we configure inertia, we can make the manipulation continue and slow down by a specific deceleration value or over a certain interval. We can configure inertia by using the ManipulationInertiaStarting event.

SUBSCRIBING TO MANIPULATIONINERTIASTARTING

ManipulationInertiaStarting is called after all of the captured manipulators have been lost and marks the start of the inertia phase of the manipulation. This event uses the ManipulationInertiaStartingEventArgs class and three supporting "behavior" classes, shown in figure 9.9.

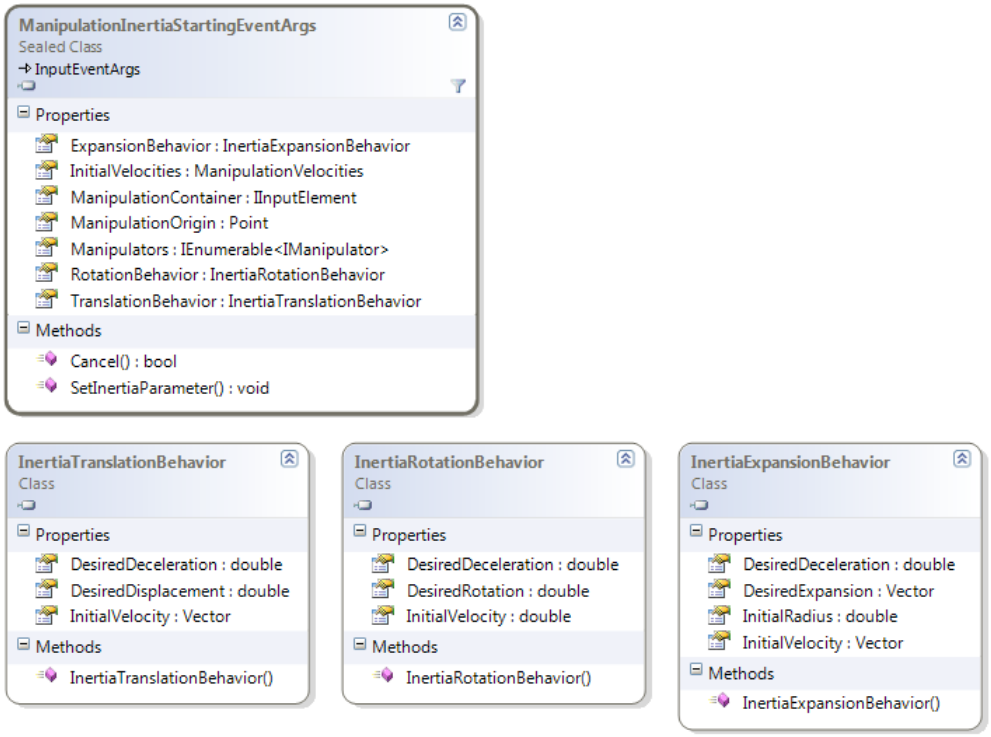


Figure 9.9 The ManipulationInertiaStarting event uses the ManipulationInertiaStartingEventArgs class which contains references to InertiaTranslationBehavior, InertiaRotationBehavior, and InertiaExpansionBehavior instances. These classes are used to configure the behavior of inertia after the manipulation phase has ended.

You can subscribe to the ManipulationInertiaStarting event similar to the other manipulation events in C# or XAML. In both cases, you'll need an event handler that looks like this:

```
void element_ManipulationInertiaStarting(object sender,
                    ManipulationInertiaStartingEventArgs e)
{
}
```

In this event handler, you can check the ManipulationContainer, ManipulationOrigin, and Manipulators, and InitialVelocities properties on the provided ManipulationInertiaStartingEventArgs instance as well as change the default behavior for TranslationBehavior, RotationBehavior, and ExpansionBehavior properties. The InitialVelocities property lets you get the velocities of the manipulation during the transition from the manipulation phase to inertia phase.

If you don't change anything, no inertia will occur. If you want to enable inertia you will need to configure one or more of the behavior properties.

CONFIGURING INERTIA BEHAVIOR

There are three types of behaviors that you can configure: TranslationBehavior, RotationBehavior, and ExpansionBehavior. Each of these can be configured independently. For example, you could configure TranslationBehavior and RotationBehavior with different deceleration values and leave ExpansionBehavior to the default of no inertia.

There are three properties that you could configure using these behavior classes: DesiredDeceleration, DesiredDisplacement, and InitialVelocity.

- **DesiredDeceleration**

This property takes a double that represents the deceleration in device-independent units per millisecond squared. This is a difficult unit to get your head around and the end result depends both upon how your specific manipulation works and the velocity when the user lifts the last finger. It's usually best to just experiment with a few values until your inertia feels right. This code snippet gives a good starting value:

```
e.TranslationBehavior.DesiredDeceleration = 0.002;
```

This will continue the manipulation and call ManipulationDelta, decelerating the translation until the delta translation has reached zero.

- **DesiredDisplacement**

The other option is to set the DesiredDisplacement to the remaining distance you want the manipulation to cover. WPF will automatically calculate the required deceleration value so that your manipulation stops exactly where you want it to. This is extremely useful for flicking items between specific preset positions or stops.

DesiredDisplacement is a magnitude and is always positive.

- **InitialVelocity**

The manipulation will continue in the direction indicated by the InitialVelocity property. Consider the following two lines of code intended to be in the ManipulationInertiaStarting event handler:

```
e.TranslationBehavior.DesiredDisplacement = 100;
e.TranslationBehavior.InitialVelocity = new Vector(1, 0);
```

These lines will cause the every manipulation to be followed by a 100 device-independent pixel (DIP) displacement in the direction of the InitialVelocity. The InitialVelocity is set to be 1 DIP per millisecond to the right.

> **NOTE**
>
> The magnitude of the InitialVelocity determines the total duration of the inertia so you may want to make sure the velocity has at least a minimum magnitude when using the DesireDisplacement method.

Do not confuse an inertia behavior's InitialVelocity property, which you can set to a new value, with the EventArg's InitialVelocities property, which is read-only. The InitialVelocity starts as the same value as the corresponding property within InitialVelocities. For example, TranslationBehavior.InitialVelocity defaults to the same value as InitialVelocities.LinearVelocity. You only need to change the InitialVelocities if you want to change the starting velocity for the inertia phase. You might set the InitialVelocity if you are using DesiredDisplacement and need to make the manipulation head in a specific direction.

You could also use the InitialVelocity to make the manipulation bounce off of boundaries. This also requires that you check for the boundaries during the ManipulationDelta and restart inertia if you reach one.

STARTING AND RESTARTING INERTIA

The ManipulationDelta event is fired during both the manipulation and inertia phases. In some cases, you may want to trigger inertia to start or restart. You can do this in the ManipulationDelta event handler by simply calling:

```
e.StartInertia();
```

This will cause the ManipulationInertiaStarting event to fire even you are already in the inertia phase.

One scenario where this would be useful is if you have a boundary check and want to prevent the manipulated object from moving out of the boundary. If the user flicks or moves the object so it would leave the boundary, call e.StartInertia() and the ManipulationInertiaStarting event will be fired. If you want the manipulation to bounce back from the boundary, you may also want to calculate and store the desired initial velocity so the ManipulationInertiaStarting event can set it.

Depending upon your application, you might want to do something slightly different depending upon whether it was in in manipulation phase or inertia phase when the boundary was reached. For example, you might want to bounce off the boundary during inertia but just complete the manipulation without

inertia if the user was still manipulating the object when it hit the boundary. You can tell whether the manipulation is in the inertia phase by checking to see if the ManipulationDeltaEventArgs.IsInertial property is true.

Calling StartInertia() always leads to a new inertia phase, but there is also a way to skip inertia and directly complete the manipulation. We will cover this in the next section.

9.3.6 Completing the manipulation

All manipulations ultimately either complete or cancel. I don't recommend using the cancel functionality, so in most cases your manipulations will always end up completing. When a manipulation has finished both the manipulation and inertia phases, WPF will fire the ManipulationCompleted event. This event uses the following event handler signature:

```
void element_ManipulationCompleted(object sender,
                                   ManipulationCompletedEventArgs e)
{ }
```

The ManipulationCompletedEventArgs gives you access to all of the same properties such as ManipulationContainer and ManipulationOrigin and also provides the FinalVelocities and TotalManipulation properties. These two properties give you some information about of the last state of the manipulation. In most cases, you will not need any of these properties and may just use the ManipulationCompleted event for cleaning up or resetting your internal logic and data for the next manipulation.

In addition to completing the manipulation naturally, you can force the manipulation to skip manipulation and jump straight to completion by calling e.Complete() in the ManipulationDelta and ManipulationStarted event handlers. The ManipulationInertiaStartingEventArgs does not have a Complete() method, but you can implicitly skip inertia to completion during the ManipulationStarting event handler by simply not configuring any of the inertia behaviors or else setting the DesiredDeceleration and DesiredDisplacement back to their defaults of double.NaN.

At this point, we have covered all of the manipulation events that occur in a normal manipulation sequence. There is one more manipulation event that we need to discuss before completing this section. Once we discuss the ManipulationBoundaryFeedback event, we will be ready to create our own custom manipulable controls.

9.3.7 Providing feedback at boundaries

We discussed one approach to handling boundaries in the inertia section: complete the manipulation if the user moves an object to the boundary. The problem with that approach is that it steals controls of the object out from under the user's fingers and the user doesn't always know why. It also requires the user to lift and replace the finger on the object to move it back in.

A better approach is to provide feedback to the user at the boundaries of a manipulation. The SurfaceScrollViewer control and controls that use SurfaceScrollViewer such as SurfaceListBox and SurfaceTextBox implement the boundary feedback feature. When you manipulate those controls past the boundary you can see the feedback it gives you.

We can add similar functionality in our controls by using the UIElement.ManipulationBoundaryFeedback event along with the ManipulationDeltaEventArgs.ReportBoundaryFeedback method.

REPORTING BOUNDARY FEEDBACK

In your ManipulationDelta event handler, you will need to have some code that determines if the object is within the boundary of a container or not. If not, you can call the ManipulationDeltaEventArgs.ReportBoundaryFeedback() method to tell it the unused portion of a manipulation. This method takes a ManipulationDelta instance, which you can create yourself. Listing 9.2 shows what a ManipulationDelta event handler that reports boundary feedback might look like.

Listing 9.2 ManipulationDelta event handler with boundary feedback

```
void element_ManipulationDelta(object sender,
                               ManipulationDeltaEventArgs e)
{
    UpdateManipulation(e.DeltaManipulation);    #A
```

```
        ManipulationDelta unused =
            GetUnusedDelta(e.DeltaManipulation);    #B

        e.ReportBoundaryFeedback(unused);           #C
    }
```

#A Updates the manipulated visual
#B Calculates excess manipulation
#C Reports feedback

This code hides some of the implementation details in helper methods. I will reveal how to actually implement the logic in a moment. For now just understand that reporting feedback is as simple as calculating the left over manipulation as a ManipulationDelta instance and then passing that that ManipulationDelta to ManipulationDeltaEventArgs.ReportBoundaryFeedback().

If the manipulation is completely within the boundaries, this code would calculate no unused manipulation and call ReportBoundaryFeedback() with a ManipulationDelta instance filled with zeros. This will have no effect, and you could add an if statement to prevent calling ReportBoundaryFeedback in that case, but the logic in the conditional might be more difficult to read than just calling it regardless.

When you call ReportBoundaryFeedback, WPF will queue up the ManipulationBoundaryFeedback event. That event is how you can get access to and take action upon this feedback, if you choose to.

ACTING UPON BOUNDARY FEEDBACK

The ManipulationBoundaryFeedback is the last of the manipulation events. It is fired only when a ManipulationDelta event handler calls the ReportBoundaryFeedback() method. This event uses the bubbling routed event strategy and fires the event on the manipulated element first. If the event is unhandled, the event bubbles up to the parent visual until it reaches the root.

If the ManipulationBoundaryFeedback event is unhandled when it reaches the Window, then Windows 7 will cause the entire Window to move in the direction of the unused manipulation. If you are using a touch screen, this should be familiar to you as it is the same thing that happens when you scroll in a browser or other application and hit the edge of the scrolling area.

Moving the entire window only makes sense in a windowed environment, though. Many multi-touch applications, especially those running on Surface hardware, run full screen. In multi-user scenarios, moving the entire window is ambiguous reaction because it isn't clear which manipulation caused it.

If you are putting in the effort to report boundary feedback in your controls, you should also handle the ManipulationBoundaryFeedback event handler and provide an appropriate feedback visual. Handling this event will prevent the entire Window from moving and also give your users a better experience.

To handle this event, you'll need to subscribe to it either at the UIElement with IsManipulationEnabled set to true or higher in the visual tree. Once you subscribe, you'll also need an event handler that looks like this:

```
void element_ManipulationBoundaryFeedback(
                        object sender,
                        ManipulationBoundaryFeedbackEventArgs e)
 { }
```

The ManipulationBoundaryFeedbackEventArgs is similar to the other manipulation EventArgs in that it provides access to the Manipulators and the ManipulationContainer as properties. It also provides the BoundaryFeedback property, which gives you access to the ManipulationDelta that the was provided to the ReportBoundaryFeedback() method.

Once you have the BoundaryFeedback information, it's really up to you to figure out what to do with it. One common approach is to apply a transformation to a visual that represents the manipulation's boundary, which may or may not be the ManipulationContainer. You could also change the visual appearance of the manipulated visual by stretching or shrinking it as appropriate. Think about the physicality and how it will integrate with inertia.

While in many cases, you may subscribe to the ManipulationBoundaryFeedback on the same control that is being manipulated, this is not always the case. If you have an ItemsControl such as ListBox and a custom ItemContainer control, it would be valid to report boundary feedback in the ItemContainer control and leave it up to the ListBox higher in the visual tree to handle it.

The best way to show how to implement the boundary feedback logic is to build a control that uses boundary feedback. In the next section, I will illustrate the design and creation of a custom manipulation control that incorporates all of manipulation techniques in this chapter. I'll also wrap up a few other applied topics about manipulations.

9.4 Applied manipulation scenarios

If you've made it this far in this chapter, then you should be fairly well versed in what the manipulation API can do. So far most of what we've covered is "book" knowledge. That knowledge is important but knowing how to use it in a practical scenario is equally important. This section is dedicated to showing you how to apply the manipulation API to real scenarios.

I'm going to cover four applied scenarios in this section. First, we will see all of the manipulation events in action by creating a reusable WPF control that uses manipulations. Next, we will see how you can create custom IManipulators for advanced scenarios. After that we will see how to manage manipulations within manipulations and then finally will discuss the subtle interaction between the raw touch API and the manipulations API.

9.4.1 Creating a reusable manipulation control

In chapter 3, we created a basic ScatterView behavior using manipulations. That involved updating a matrix using the ManipulationDelta data. Rather than rehash that example, let's look at an actual control that illustrates all of the manipulation API concepts and techniques we've seen so far.

For our example control, we will build a custom slider. This slider has a "thumb" consisting of rectangle that you can manipulation horizontally within a long container, represented by another rectangle. We'll refer to the distance that the thumb can travel within the container as the manipulation range. Figure 9.10 shows what this control looks like.

Figure 9.10 A custom slider control that will illustrate boundary feedback. The dark gray rectangle is the thumb and can move horizontally within the light gray rectangle.

We can build this visual pretty easily. This control is already built out as the Boundary1DSlider user control in the BoundarySlider project. Listing 9.3 shows the XAML for this control, taken from Boundary1DSlider.xaml.

Listing 9.3 XAML for the Boundary1DSlider control

```
<Border Name="container"                              #1
        Background="LightGray"
        Width="600"
        Height="100">
    <Border.RenderTransform>
        <TranslateTransform
            x:Name="containerTranslate" />            #2
    </Border.RenderTransform>
    <Border Name="thumb"                              #3
            Background="DarkGray"
            IsManipulationEnabled="True"
            HorizontalAlignment="Left"
            VerticalAlignment="Center"
            ManipulationStarting="thumb_ManipulationStarting"
            ManipulationDelta="thumb_ManipulationDelta"
            ManipulationInertiaStarting=
```

```
                [CA]"thumb_ManipulationInertiaStarting"
            ManipulationCompleted="thumb_ManipulationCompleted"
            ManipulationBoundaryFeedback=
                [CA]"thumb_ManipulationBoundaryFeedback"
            Width="200"
            Height="80">
        <Border.RenderTransform>
            <TranslateTransform
                x:Name="thumbTranslate" />                    #4
        </Border.RenderTransform>
        <s:TouchVisualizer.Adapter>
            <s:TouchVisualizerRectangleAdapter />             #5
        </s:TouchVisualizer.Adapter>
    </Border>
</Border>
```

#1 Container for the slider
#2 Translates container for boundary feedback
#3 Thumb for manipulation
#4 Translates thumb within container
#5 Enables visualization trails

Cueballs in code and text below

At the core this control is just a Border #3 within a Border #1. Each Border has its RenderTransform set to a named TranslateTransform. The thumb's TranslateTransform #4 moves the thumb within the container. The container also has a TranslateTransform #2 that will be used for the boundary feedback. Since the thumb is within the container, any translations applied to the container TranslateTransform #2 will also move the thumb.

Notice that the HorizontalAlignment for the Thumb is set to Left. This means that a translation of 0 causes the thumb to align with the left boundary and a translation of 400 (container.ActualWidth - thumb.ActualWidth) causes the thumb to align with the right boundary.

We have also set a TouchVisualizerRectangleAdapter #5 for the thumb. This will enable visualization trails between touches and the thumb when the touch leaves the boundary of the thumb.

The thumb #3 has manipulations enabled and is subscribed to all of the manipulation events except for the rarely used ManipulationStarted.

In the ManipulationStarting event handler, we configure the ManipulationContainer to be the container. In the ManipulationDelta event handler, we will indirectly update the thumb's TranslateTransform.X property using the e.DeltaManipulation.Translation.X value. We don't want to do this directly though otherwise we cannot calculate the values needed for our desired the boundary feedback behavior. To see how to make the ManipulationDelta event handler work right, we need to use a helper class that I have made.

MANIPULATIONDELTA AND BOUNDEDMANIPULATIONRANGE

For any control you create, you will need to design a behavior for what happens when the user tries to manipulate it beyond the boundaries. For the Boundary1DSlider control, I have chosen to emulate the stretchy feedback used by windows. That behavior will let you stretch the container from its default position when the thumb hits an edge but only so far. I have created a helper class called BoundedManipulationRange that helps to create this behavior.

BoundedManipulationRange tracks a Position value within a range and provides a BoundedPosition that is limited to LowerBoundary and UpperBoundary. It also calculates an ElasticOffset if the Position is outside of the boundaries that is progressively slows down so it never exceeds the boundary plus an ElasticMargin. Figure 9.11 illustrates this concept.

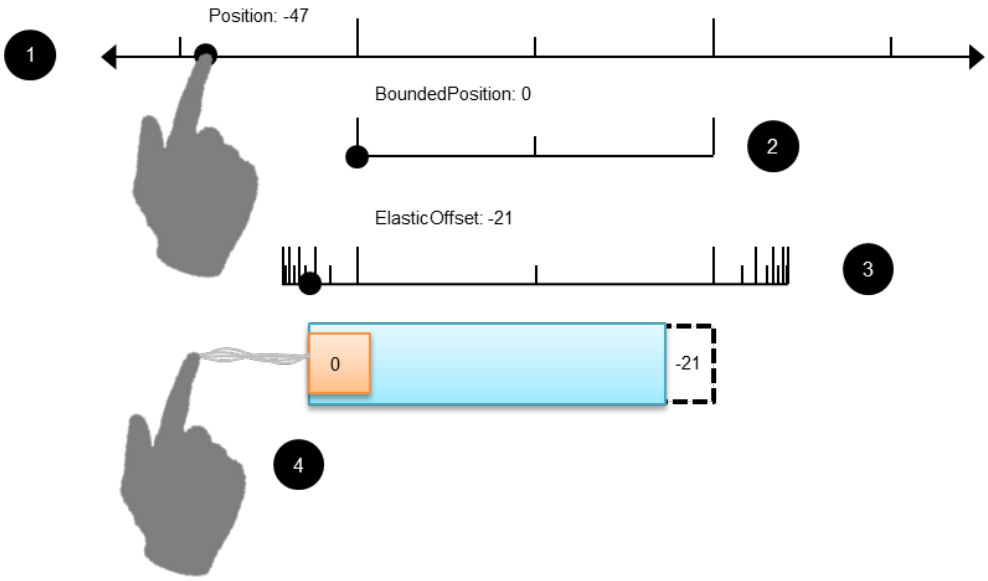


Figure 9.11 The user has manipulated to the -47 Position #1. This is outside the LowerBoundary of 0 and the thumb is translated to the BoundedPosition #2 of 0. The Container is translated by the ElasticOffset #3 of -21, providing feedback on exceeding the boundary, and the TouchVisualizer draws a tether #4 between the touch and the captured UIElement. Even if the user translated to -600, the ElasticOffset will never exceed the ElasticMargin, which defaults to 40.

BoundedManipulationRange is found within the Blake.NUI project in the Blake.NUI.WPF.ManipulationHelpers namespace. If this were a trivial example, we might directly update the TranslateTransform like this:

```
thumbTranslate.X += e.DeltaManipulation.Translation.X;
```

Instead of doing that, we will add a BoundedManipulationRange field called horizontalRange to the Boundary1DSlider and initialize it in the constructor:

```
horizontalRange = new BoundedManipulationRange()
    {
        ElasticMargin = 40,
        Position = thumbTranslate.X
    };
```

This configures the initial position and how far we want the boundaries to stretch. We will primarily use this class in the ManipulationDelta event handler. First we want to update the boundaries:

```
horizontalRange.LowerBoundary = 0;
horizontalRange.UpperBoundary = container.ActualWidth - thumb.ActualWidth;
```

We update the boundaries each ManipulationDelta because it is possible for the widths to change due to other events in the interface. Finally we can update the thumb's TranslateTransform indirectly:

```
horizontalRange.Position += e.DeltaManipulation.Translation.X;
thumbTranslate.X = horizontalRange.BoundedPosition;
```

What we're doing here is adding the DeltaManipulation to an unbounded BoundedManipulationRange.Position property. When we access the BoundedManipulationRange.BoundedPosition, it will clamp the Position to be within the LowerBoundary and UpperBoundary properties.

When the user moves the thumb beyond the boundaries, the thumb will stay within the boundaries because it is using the BoundedPosition. The Position property may travel beyond the boundaries, though. We want to allow this so that we can properly implement the stretchy boundary feedback behavior. All we need to do to calculate and report the boundary feedback is use the BoundedManipulationRange.ElasticOffset property:

```
e.ReportBoundaryFeedback(new ManipulationDelta(
                            new Vector(horizontalRange.ElasticOffset,
                                       0),
                            0, new Vector(), new Vector()));
```

When we access the ElasticOffset property, it first calculates how far the Position value is outside of the boundaries, if at all, and then applies a stretchy function that normalizes the offset to be within the ElasticMargin value.

After ReportBoundaryFeedback is called, the ManipulationBoundaryFeedback event is fired on the thumb. Here is what that event handler contains:

```
Vector delta = e.BoundaryFeedback.Translation;
containerTranslate.X = delta.X;
containerTranslate.Y = delta.Y;
```

This updates the container's TranslateTransform, moving the thumb along with it. The values here are the ones provided by the ElasticOffset property earlier. Overall this provides a smooth transition between directly manipulating the thumb and limiting the manipulation range. This behavior tells the user "yes, I know you're trying to go farther but I'm stretching my limits already."

The next thing we need to discuss is how inertia is used in this control.

CONFIGURING INERTIA

There are two inertia conditions that we need to account for:

1. Inertia starts when the Position is within the range
2. Inertia starts or restarts when the Position is out of bounds

In the first case, we just want to set a DesiredDeceleration to allow the thumb to travel a bit before stopping. In the second case, we want the inertia to bring the Position back in bounds, which will reset the container back towards its natural zero position. To put it another way, when the user stretches the thumb and container past the boundary and then releases, we want it to snap back to resting position.

We can use two BoundedManipulationRange properties to help with this. The IsPositionOutOfBounds property tells us whether the Position value is out of the boundaries and the BoundaryOverflow property tells us how far the Position is from the boundary. Listing 9.4 shows the code in the ManipulationInertiaStarting event handler.

Listing 9.4 Boundary1DSlider's ManipulationInertiaStarting event handler

```
if (horizontalRange.IsPositionOutOfBounds)
{
    double displacement = Math.Abs(horizontalRange.BoundaryOverflow) + 1;
    e.TranslationBehavior.DesiredDisplacement = displacement;

    Vector velocity = new Vector();
    velocity.X = -Math.Min(2,displacement/10) *
                  Math.Sign(horizontalRange.BoundaryOverflow);
    e.TranslationBehavior.InitialVelocity = velocity;
}
else
{
    e.TranslationBehavior.DesiredDeceleration = 0.002;
}
```

In this code, if the position is out of bounds, we calculate a displacement large enough to bring it back into bounds and a velocity that will move in the correct direction. The `Math.Min(2,displacement/10)` prevents the velocity from being too large if the displacement is smaller than 20. If the velocity and displacement were both 2, for example, the inertia would instantly complete. If the Position is in bounds, we simply set a regular displacement.

We also need to account for what happens if the user flicks the slider with a large velocity. If it is large enough, the position will travel way beyond the boundaries during inertia phase and may take a second or two. During this time the thumb will just sit at the edge and the container will be slightly displaced. In order to prevent this delay, I have added the code in listing 9.4 to the ManipulationDelta event handler.

Listing 9.4 ManipulationDelta logic to restart inertia if inertia travels past boundary

```
if (e.IsInertial)
{
    if (horizontalRange.IsPositionBelowLowerElasticMargin &&
        e.Velocities.LinearVelocity.X < 0)
```

```
        {
            e.StartInertia();
        }
        else if (horizontalRange.IsPositionAboveUpperElasticMargin &&
                 e.Velocities.LinearVelocity.X > 0)
        {
            e.StartInertia();
        }
    }
```

All this code does is check if during an inertia phase the position has travelled past the elastic margin on the lower bounds or the upper bounds. If so, it will cause inertia to restart, allowing the ManipulationInertiaStarting event handler to calculate a new displacement.

Finally, I have added a small bit of logic in the ManipulationCompleted event handler:

```
if (horizontalRange.IsPositionOutOfBounds)
{
    InertiaDummyManipulator.ManipulateAndStartInertia(thumb);
}
```

This code again checks if the Position is out of bounds and if so, uses the another Blake.NUI helper class to restart inertia. The InertiaDummyManipulator class simply adds a dummy manipulator to the provided UIElement and then immediately removes it. This lets us initiate inertia to even if the element is not being manipulated.

We want to start inertia again in this case because the manipulation completed with the Position out of bounds. This might happen if the user has flicked the thumb towards a boundary but the logic in listing 9.4 has not been triggered because the Position didn't go far enough out of bounds.

The BoundarySlider project also has a Boundary2DSlider which uses all of the same logic but adds another BoundedManipulationRange to track vertical movement.

SLIDEOUTCONTROL BONUS EXAMPLE

In addition to the BoundarySlider controls we just discussed, I've also built a more comprehensive and reusable control that illustrates best practices for control design. This control is called SlideOutControl and is derived from HeaderedContentControl. It displays the header as a handle and hides the content. You can touch header and slide it out to reveal the content. This additional content could be text, menu options, or anything that you might want to display. Figure 9.12 shows the SlideOutControl in collapsed and expanded form.

Figure 9.12 On the left the SlideOutControl is collapsed. In the middle the control is being pulled down and eventually reaches the fully extended position shown on the right.

While the control is shown here all alone against a white background, it would make sense to line it up against the edge of a screen or the edge on another visual to reinforce the metaphor of sliding out content from under something else.

One key difference between BoundarySlider controls and the SlideOutControl is the BoundarySliders are user controls while SlideOutControl is a custom control.

User controls and custom controls

There are two ways to build WPF controls: user controls and custom controls. User controls are convenient because they have associated XAML files and the XAML tree is automatically wired up and exposed to the class. The drawback of user controls is that they cannot be extended or retemplated easily. The best approach is to use user controls for composing together other controls but use custom controls to encapsulate reusable units of interface visuals and behavior.

Custom controls are not automatically associated with XAML files in the same way that user controls are. If a custom control, such as SlideOutControl, needs a XAML template, it sets the DefaultStyleKeyProperty metadata. This will cause WPF to look for a XAML style that targets the same type as the control. When it finds one, it will apply the style, including any enclosed control templates, to the control. The control's code can get access to the visual tree in a control template by overriding the OnApplyTemplate() method and using the GetTemplateChild() method.

This seems more complicated but it is the recommended approach for any control you build that may be reused, especially if developers using the control may create a custom control template or derive from the control and override methods or change behaviors. Regular user controls do not support those scenarios well.

Since SlideOutControl is a custom control, it can easily be retemplated by simply associating it with a new control template that has the template parts it is expecting. SlideOutControl encapsulates an interface behavior that can be reused with any content that you put in it as well as any custom control templates or visual styles you desire.

In the next section we will see a slightly more advanced example of custom controls that use manipulations and see how to handle manipulations nested within manipulations. This will lead us to the final topic of the chapter, filtering manipulators.

9.4.2 Nesting manipulations

As you create more complex interfaces, you'll sometimes find a need to have controls with manipulations inside of other controls with manipulations. This sometimes poses a few challenges but they are easily overcome. To illustrate another manipulable control that can be composed, I have created the NestedManipulations sample application.

The NestedManipulations application includes a control called RectangleManipulationContentControl. This control illustrates the use of most of the manipulations API as well as the boundary feedback feature we just learned in section 9.4.1. Figure 9.13 shows a screenshot of the NestedManipulations interface.

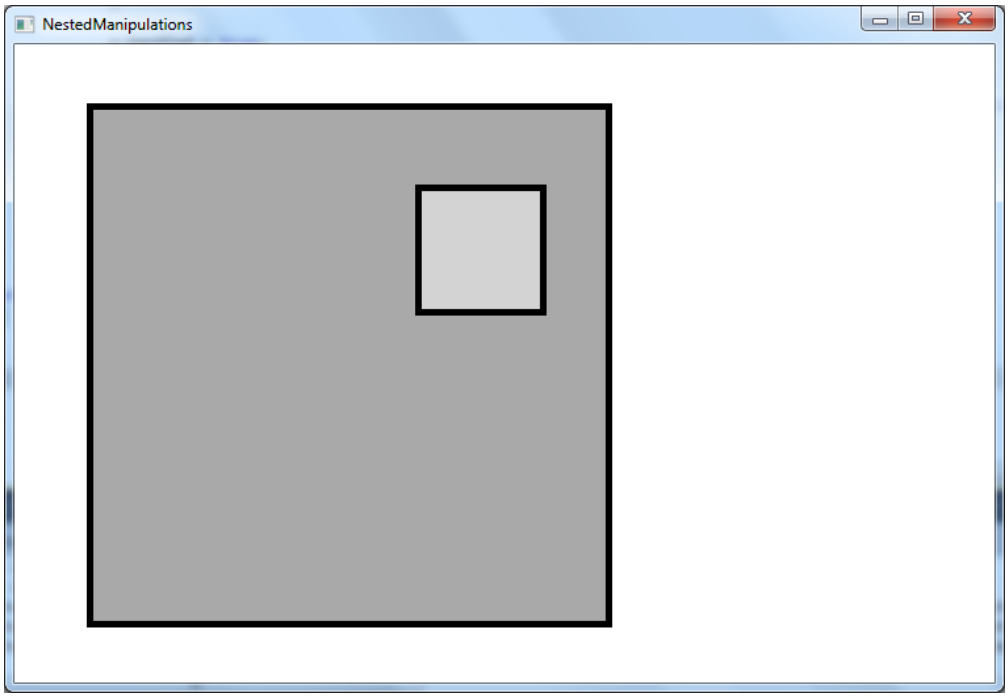


Figure 9.13 The Nested Manipulations sample lets you manipulate a small box within a large box, and manipulate the large box within the window. The layout includes an inner RectangleManipulationContentControl inside an outer RectangleManipulationContentControl. Both controls can move freely within their containers.

The RectangleManipulationContentControl can be translated freely within its container. It is derived from ContentControl, so you can embed other content within it. In this example I have a smaller RectangleManipulationContentControl nested within a larger RectangleManipulationContentControl.

At this point, I'd encourage you to try out this sample application. Move each control around and see what happens when you reach boundaries. Also notice that you can manipulate both the inner and outer at the same time in different directions, although go too far and you'll reach a boundary. Finally, try flicking the outer and then the inner borders to see inertia in action.

Here are two tips that may be helpful when dealing with nested manipulations.

PROTECT YOUR EVENT HANDLERS

Keep in mind that the manipulation events will bubble up if you do not handle them. If you have a manipulation operating within another manipulation and forget to set e.Handled to true then you might see some really odd behavior that is rather difficult to debug.

On the other side of things, you should also check at the beginning of your manipulation event handlers to ensure that the UIElement that your event is processing is the one you expect. If someone else has some manipulation code that ends up nested inside of your control and they forget to handle their events, it can mess up your control. You can prevent this by checking whether e.OriginalSource is the UIElement that you are expecting it to be.

You should get in the habit of adding these three lines at the beginning of your manipulation event handlers, and perhaps other input event handlers as well:

```
if (e.OriginalSource != MyManipulationElement)
    return;
e.Handled = true;
```

This will make your control a good citizen within the visual tree and also protect against other controls that are not written as well as your controls.

In fact, while preparing the NestedManipulations application, I had forgotten to add this check for the ManipulationInertiaStarting event handler and the manipulations were going crazy but not in a way that

made it obvious what the problem was. Once I double-checked and followed my own advice, everything started working smoothly again.

NOTIFY CHILD VISUALS OF MANIPULATION DELTA

Suppose that you have manipulable content in a panning background container and when you pan the background all of the content moves as well. This is very similar to several real life container situations and users will expect it to work like real life. In real life the user can move the container around and also hold objects within the container to prevent them from moving.

The NestedManipulations applications has this feature. Touch and move the outer control and notice that the inner control moves with it. Now keep moving the outer control but with your other hand touch, hold, and move the inner control. Everything works intuitively here, but it required a little work in the code. Try the same test but this time comment out this line at the end of the border_ManipulationDelta() method in RectangleManipulationContentControl.cs:

```
NotifyChildManipulationDelta(e);
```

That line causes the control to notify child content of each ManipulationDelta. Without the line, the inner control won't stay still if you are holding it while moving the outer control.

What happens in the NotifyChildManipulationDelta method is this code checks to see if the child content implements the INotifyParentManipulationDelta interface, and if so calls the sole interface method ParentManipulationDelta(). The child code will then check to see if the manipulation is active, and if so will add the opposite of the parent's manipulation to the child's manipulation. If you are touching the child control, the parent's manipulation will be cancelled out, letting you hold the child in place even though the container is moving.

The specific implementation of this feature is rather sparse in this sample. You might need to build something a little robust if you implement this for production code, but hopefully this should give you a good start.

In the next and final section of this chapter, we will see how the raw touch and manipulations APIs work together and how you can use that knowledge to filter which TouchDevices are used in a manipulation.

9.4.3 Filtering manipulators

Depending upon the input hardware you are using, TouchDevices may have some extra hidden information that you need to access in order to create a great user experience. For example, the Surface SDK can give you extra information about a TouchDevice using the TouchExtensions in Microsoft.Surface.Presentation.Input namespace:

```
TouchDevice device = GetTouchDevice();
bool isFinger = device.GetIsFingerRecognized();
bool isTag = device.GetIsTagRecognized();
```

One way to use this information is to filter which TouchDevices participate in which manipulations. You could configure the interface so that some manipulations ignore TouchDevices that are really tags and not fingers.

One scenario where this might be useful is on an infinite canvas with several items covering much of the foreground. One valid design would be to allow fingers to manipulate the foreground items but allow physical objects marked with tags control the background, even if the objects were placed over a foreground item. This scenario would allow users to pan or zoom the background canvas even when a foreground item takes up the entire screen.

This scenario involves nested manipulations and requires us to filter which TouchDevices are allowed to participate in foreground object manipulation or background canvas manipulation. In order to make this work, we need to understand the relationship between the raw touch and manipulations API.

THE RELATIONSHIP BETWEEN RAW TOUCH AND MANIPULATIONS

So far we have talked about the raw touch API and manipulations API separately. The two magically work independently, but we don't know yet how they work together. There are really only three things that you need to know.

First, the raw touch events are always fired first, and they always bubble up to the root before any manipulation logic occurs. This means that if you subscribe to TouchMove and ManipulationDelta and set breakpoints, you'll always hit the TouchMove's breakpoint before you hit the ManipulationDelta breakpoint. You can also subscribe to TouchMove all the way up the visual tree to the window and verify that each of those are fired before ManipulationDelta occurs.

NOTE

Recall that the TouchDown, TouchMove, and TouchUp events use the bubbling strategy and are fired starting at the target element and bubbling up to the root element. These three events also have the tunneling variants PreviewTouchDown, PreviewTouchMove, and PreviewTouchUp. For brevity, I will not mention the Preview events when discussing the relationship with the manipulation API, but they still occur and could be handled regardless.

Second, there is a little dance that occurs between several events when you touch a UIElement with IsManipulationEnabled set to true. Applying what I said in the last paragraph, the TouchDown event is fired and bubbles up to the root element. If the TouchDown event is handled, then nothing else happens. If the TouchDown event is not handled, then WPF will call into the manipulation logic. Figure 9.14 illustrates this dance.

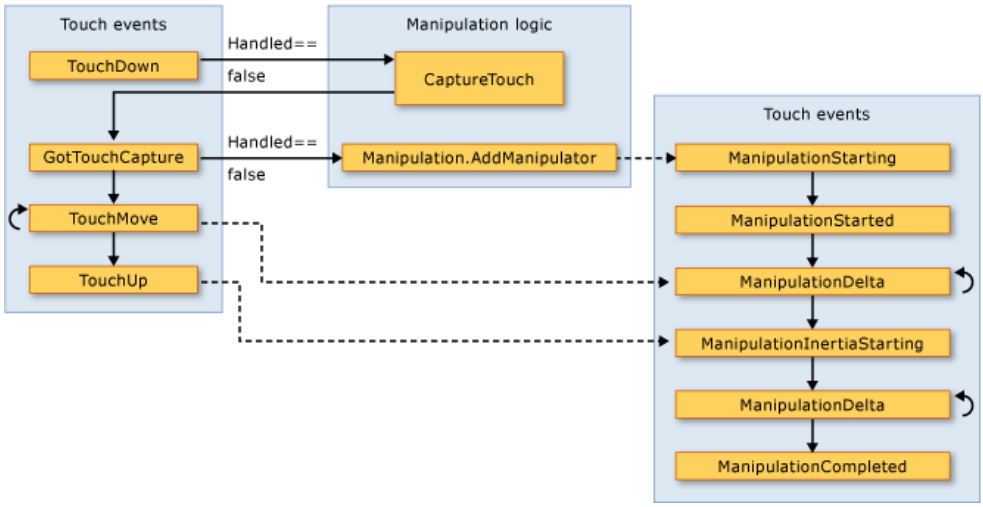


Figure 9.14 Flow diagram of touch events, manipulation events, and manipulation logic.

This manipulation logic will search the visual tree starting at the TouchDown event's target element and moving upwards towards the root element. It is looking for any UIElement that has IsManipulationEnabled set to true. If doesn't find one, nothing happens. If it does, it will stop at the first UIElement that is manipulable and capture the TouchDevice to that UIElement. This sets off another raw touch event: GotTouchCapture.

If GotTouchCapture is handled or if the capture is changed or released, nothing else happens in the manipulation logic for this event sequence. If GotTouchCapture is not handled, then the manipulation logic is confident that it is allowed to add this TouchDevice to a manipulation. It will create a manipulation if none exists, or add the TouchDevice to an existing manipulation on the UIElement in question.

NOTE

TouchDevice derives from InputDevice and also implements the IManipulator interface.

From this point, all of the standard manipulation events occur. The ManipulationDelta event is preceded by one or more TouchMove events, and ManipulationInertiaStarting occurs after the last IManipulator (typically a TouchDevice) leaves the manipulation, so is typically preceded by a TouchUp event.

ManipulationDelta doesn't care whether the TouchMove event was handled or not. Likewise, ManipulationInertiaStarting is not affected by handling the TouchUp events. Only the initial dance between the TouchDown and GotTouchCapture events and the manipulation logic are affected by handling the events. Setting e.Handled to true for either of these events will prevent the TouchDevice from participating in a manipulation.

I mentioned there are three things you need to know about how raw touch works with manipulations. The first was that the touch events are fired first and travel all the way to the root, and the second was that handling the TouchMove or GotTouchCapture events will block a TouchDevice from being added to a manipulation.

The third thing is a little trick that exploits the fact that the manipulation logic is just a state machine. Remember that if TouchMove is unhandled, the WPF manipulation logic looks for the first manipulable UIElement it can find. You can trick it into using a different manipulable UIElement farther up the visual tree. All you have to do is in one of the TouchMove event handlers capture the TouchDevice to another UIElement that has IsManipulationEnabled set to true:

```
void Border_TouchDown(object sender, TouchEventArgs e)
{
    e.TouchDevice.Capture(border);
}
```

This will immediately cause the GotTouchCapture event to fire on that element and then we skip into the second phase of the manipulation logic that occurs when GotTouchCapture is unhandled. The difference is now you have told WPF to add the TouchDevice to a specific manipulation.

NOTE

This trick only works in the direct hierarchy of the TouchDown event and only for UIElements that have IsManipulationEnabled set to true. You cannot add the TouchDevice to a manipulation across the visual hierarchy on an element that TouchDown would not be fired on. The capture trick only works for direct parents or children of the manipulable element.

Now that you know how the raw touch and manipulations events work together, we can solve the original challenge of filtering manipulators. There are two techniques that we can use to filter the manipulators. I like to describe them as opt-out manipulator filtering and opt-in manipulator filtering.

OPT-OUT MANIPULATOR FILTERING

The scenario for opt-out filtering is that you have a manipulable UIElement and you want to prevent certain TouchDevices from participating in the manipulation. If you do nothing, the TouchDevices would be added to the manipulation, but if you take the following action, they will be filtered out, hence the name opt-out.

Add a GotTouchCapture event handler to the manipulable UIElement and check the properties on e.TouchDevice. If the TouchDevice should be in the manipulation, do nothing and return. If the TouchDevice should not be in the manipulation, then you will want to capture the TouchDevice to the next parent in the visual tree that has IsManipulationEnabled set to true. If there are no other manipulable UIElements, then you can just release the capture by passing null to TouchDevice.Capture().

The result of this is that the TouchDevice will pass through to the next level as if the current UIElement did not have manipulations enabled in the first place. I've written a helper method that implements this logic and makes it easy to implement opt-out filtering. It is located in the Blake.NUI project in the Blake.NUI.WPF.Utility.ManipulationUtility class.

Listing 9.5 shows how the NestedManipulations application uses this helper to prevent tags from manipulating the RectangleManipulationContentControl. The IgnoreTags property lets me configure the inner control to ignore tags and let the outer control still respond to them.

Listing 9.5 Filtering out tags and passing the capture up the visual tree

```
void border_GotTouchCapture(object sender, TouchEventArgs e)
{
    if (e.TouchDevice.GetIsTagRecognized() && IgnoreTags)
    {
        ManipulationUtility.CaptureTouchDeviceToManipulationEnabledParent(
            this,
            e.TouchDevice);
    }
}
```

Opt-out manipulation filtering lets you tell WPF to ignore certain TouchDevices and pass them up the visual tree for other potential manipulations.

OPT-IN MANIPULATOR FILTERING

Opt-in filtering could also be known as greedy manipulations. This is because you can use it to force TouchDevices to be added to a specific UIElement's manipulation even if a child manipulation would have otherwise captured it.

The opt-in technique is really simple. Actually we already covered most of it. All you need to do is add a TouchDown event handler check the TouchDevice. If the UIElement should be greedy about this TouchDevice then you should capture it to the manipulable UIElement:

```
e.TouchDevice.Capture(Border);
```

This will prevent the manipulation logic from even looking for manipulable elements and instead the TouchDevice will be directly added to the greedy UIElement's manipulation.

If you control the entire visual tree then you could use either opt-out or opt-in manipulation filtering techniques to achieve the same effect. In most cases, opt-out would be cleaner. If you do not control the entire visual tree, such as if you are creating a content control or items control and can't know what the children might be, then you may need to use opt-in manipulation filtering to implement a specific user experience.

Now that we have discussed opt-out and opt-in manipulation filtering, we have covered the entire manipulation API as well as several intermediate to advanced practical scenarios for using manipulations. This knowledge should give you a really great boost in your ability to create custom, advanced natural user interfaces.

9.6 Summary

We covered a lot in this chapter. We started with manipulation concepts, learned about how the manipulation events worked, and finished up with a few advanced manipulation scenarios.

A lot of the content in this chapter is not documented anywhere else and I believe that this knowledge will be useful for developers who are creating amazing interfaces of the future. Here are some tips to remember:

- Avoid feedback loops by setting the ManipulationContainer in the ManipulationStarting event handler to a UIElement that is not affected by the current manipulation
- Configure the Pivot property in the ManipulationStarting event handler to enable single finger rotation
- No inertia will occur unless you set the DesiredDeceleration or DesiredDisplacement in the ManipulationInertiaStarting event handler
- During the ManipulationDelta event handler, you can check the e.IsInertial to see if inertia is active and can restart Inertia by calling e.StartInertia()
- Use ReportBoundaryFeedback and the ManipulationBoundaryFeedback event to provide a better experience when the user reaches the limits of a manipulation
- Use the BoundedManipulationRange helper from Blake.NUI to easily calculate bounded manipulations and report feedback
- Always handle your manipulation event handlers and code defensively to protect against child controls that did not handle their manipulation events

- Use the opt-out and opt-in techniques for filtering which TouchDevices participate in which manipulations within the visual tree

The next chapter, chapter 10, builds upon the knowledge of both the raw touch and manipulations APIs to let you drag-and-drop objects within and between containers and also teaches you about the customizing touch visualizations. Chapter 10 wraps up this Part 2 of this book.

10 Integrating Surface frameworks

In chapters 8 and 9, we learned about the raw touch and manipulations APIs built in to WPF 4. You can build anything you can imagine with those APIs, but some scenarios will take longer than others. Surface SDK 2.0 makes the developer's life easier, again, by making it easy to address two of the more difficult challenges faced by almost all practical multi-touch applications.

The first challenge is providing immediate feedback to user input. Per-finger feedback is necessary for a good user experience and in fact has been shown to improve the user's accuracy. Surface SDK solves this with the Touch Visualization Framework that creates per-touch visuals automatically increase the usability of your applications.

The second challenge is multi-touch drag-and-drop. The native WPF drag-and-drop works with the mouse only and does not work well in multi-touch scenarios. Surface SDK provides a Drag-and-Drop Framework that is designed specifically for multi-touch and provides a rich set of interactions.

This chapter will cover how to use, configure, and extend Surface SDK's Touch Visualization Framework and Drag-and-Drop Framework.

10.1 Visualizing touches

The Surface SDK includes the Touch Visualization Framework. This framework displays small ethereal visualizations for every TouchDevice. The visualizations are useful because they give the user immediate feedback and confirmation that the system is responsive to input, even if the touched visual elements are not interactive.

If you've been following along the code samples in previous chapters, you have already seen the Surface SDK touch visualizations. They are already integrated into the SurfaceWindow's control template. The Touch Visualization Framework allows you to customize and extend the behavior of the touch visualizations.

In this section, we will learn about the TouchVisualizer control, how to configure the colors and behavior of the visualizations, and how to customize the behavior for your own controls.

10.1.1 Understanding the TouchVisualizer

At the core of the Touch Visualization Framework is the TouchVisualizer control. The TouchVisualizer control responds to raw touch events and displays small animated visuals that correspond to each TouchDevice. It is aware of when the TouchDevices are captured to controls and modifies the visual to help the user understand how the touches will interact with the interface.

BENEFITS OF TOUCH VISUALIZATIONS

One of the challenges with touch, and input in general, is making the user aware of what will happen if they interact with the interface. The interaction may be entirely different depending upon the current and previous positions of a finger.

For example, a button will respond to a finger that is placed directly onto the button but not respond to a finger that is placed outside of the button and then slid over the button, even though the two fingers are in the same position. To give another example, if the user scales a ScatterViewItem to its maximum size but keeps going, the user will still be able to move and rotate the SVI even though the fingers are not touching it.

In both of those examples, the interface has internal states that drive the interaction. Without visual cues to reveal the hidden states, users can become disoriented or confused. See the sidebar for additional discussion.

Unexpected behavior and sources of error

There are many subtle design decisions that went into the creation of the Surface controls and the touch visualizations. Some of these decisions are described in a paper written by Wigdor et. al. titled "Ripples: Utilizing Per-Contact Visualizations to Improve User Interaction with Touch Displays" and published in the proceedings of UIST 2009. This paper describes and shows various design decisions that went into the Surface controls and touch visualizations.

In its discussion, the paper describes several sources of error that cause unexpected behavior in certain touch systems. In order to clarify the discussion of certain control behaviors, I would like to describe some of these common errors.

Accidental activation - This refers to unintended changes in the application in response to either an intentional touch or movement being misinterpreted or from an unintentional touch, e.g. from a palm or sleeve. Users can be confused by the cause of these changes. Touch systems should help the user identify the source of accidental activation so they can correct their behavior

Multi-capture - Unlike mouse systems where controls can only capture the one mouse cursor, multi-touch systems can have controls that capture more than one touch input. A control such as the ScrollViewer has two potential degrees of freedom, horizontal and vertical. Each touch has at least two degrees of freedom. If two or more fingers touch a ScrollViewer, there can be confusion about how the four or more degrees of input freedom map to the two degrees of freedom of the control. Touch systems should provide appropriate behavior and feedback in this situation.

Physical manipulation constraints - When a control reaches the limit of manipulation, such as a boundary or maximum size, the user can no longer directly interact with it. Touch systems should provide appropriate behavior and feedback when constraints are reached.

Interaction at a distance - When a touch is captured to a control, such as a vertical slider, the touch can still change the vertical position of the slider even if it moves outside the horizontal bounds of the control. This interaction at a distance can potentially cause confusion. Touch systems should provide a way for users to understand when a touch may interact at a distance.

Stolen capture - When you combine multi-capture and interaction at a distance, you may have situations where a finger has triggered a state, such as a button press, that is retained while the finger outside the control and prevents another finger that is inside the control from changing the state. This may be an unexpected behavior, particularly if the two fingers belong to different users. Touch systems should provide feedback in cases of stolen capture.

These are only a subset of the topics covered in the Ripples paper. I recommend reading the full paper, which can be found here: http://www.wigdor.com/daniel/research/UIST2009_Ripples.pdf

The TouchVisualizer control helps to minimize user confusion by giving them visual cues when the interface may behave in a consistent but non-intuitive manner. There are a few different parts of the visualization that help illuminate different aspects of the interaction.

PARTS OF THE VISUALIZATION

The touch visualizations minimize several ambiguities in the interface interaction and so have a few parts. Figure 10.1 shows a composite of the different parts of the visualization.

Figure 10.1 Touch visualization parts, from top to bottom: a glow displayed under a motionless or slowly moving finger, a trail following a quickly moving finger, a glow with an expanding ring displayed at the point of TouchDown, and a glow with a tether connecting it to the rectangle that captured the TouchDevice. The ring is displayed at initial TouchDown position and not moved with the TouchDevice, which explains why the moving glow is not centered within the ring.

Table 10.1 expands upon the role of each part of the touch visualization.

Table 10.1 The touch visualizations consist of several parts with different behaviors

Part	Description
Glow	A semi-transparent ellipse centered on the touch position when the touch is not moving or moving slowly
Ripple	An animated ring displayed on TouchDown. If the TouchDevice is captured, the ring animates out to in, otherwise in to out.
Trail	When the TouchDevice is moving quickly, the glow is replaced by a thin short line trailing the position of the TouchDevice. This reduces the perception of input lag.
Tether	An organic, animated curve that connects the TouchDevice with the UIElement that has captured it. Normally only drawn when the TouchDevice has left the boundary of the UIElement or when the number of inputs has exceeded the number of degrees of freedom.

Now that we have learned about the benefits of touch visualizations and what they look like, let's move on to see how we can actually use and configure touch visualizations.

10.1.2 Configuring touch visualizations

The Touch Visualization Framework is easy to use and allows you to customize the visualization behavior in a variety of ways. In this section we're going to learn how to include the TouchVisualizer control in your window, how to disable visualizations per element, and how to change the colors of the visualizations. We will also see how to further customize the behavior for specific visual elements by using TouchVisualizerAdapters.

ADDING THE TOUCHVISUALIZER

Each window should have one and only one TouchVisualizer control. You normally do not ever need to worry about adding it because the TouchVisualizer is included in the control template of SurfaceWindow.

If you are not using SurfaceWindow, or need to customize the window control template, you can still add the TouchVisualizer to your visual tree. You would need to follow two guidelines to make sure it renders properly.

First, you should place the TouchVisualizer as the top-most element in the window. For example, if the root panel is a Grid, then place the TouchVisualizer as the last child of the Grid to ensure it is rendered on top of all of the other content.

Second, the TouchVisualizer should fill the entire window without any transformation or margin. If it does not, it will not render any visualizations.

The TouchVisualizer will register itself for touch events and display the visualizations as appropriate. We can configure how touches are visualized when the user touches our controls by setting attached properties or calling static methods on the TouchVisualizer class.

There are three aspects of the visualization that we can configure: whether to show a visualization, the color of the visualization, and how tethers are displayed.

DISABLING VISUALIZATIONS

You can disable the visualizations by settings the TouchVisualizer.ShowsVisualization attached property. This property defaults to true, but when set to false then a TouchDevice will not be visualized when the TouchDevice's TouchDown event occurs within the specified visual. In XAML, you can add this directly to any UIElement:

```
<Border s:TouchVisualizer.ShowsVisualizations="False">
    <!-- Content -->
</Border>
```

You can also configure this in code:

```
UIElement element = GetElement();
TouchVisualizer.SetShowsVisualizations(element, false);
```

In most cases, you will want to let visualizations default to on. When they are on, you can also configure the colors that are used in the visualization.

CHANGING COLORS

Touch Visualization has three configurable colors. The default values work well for almost any use, but you may want to customize the colors in certain ways. You might want to change the colors for an entire window to better match your color palette or to comply with a customer's branding guidelines. You could also change the colors for a specific control to highlight interaction with that control.

The three colors are known simply by number: 1, 2, or 3, but they serve individual roles within the visualization. Table 10.2 enumerates the default values of the colors and what the color is used for.

Table 10.2 The three touch visualization colors serve different purposes within the visual

Color number	Default color	Default color (hex)	Used in…
1	White, 40% transparent	#66FFFFFF	Touch glow inner gradient, tether secondary
2	Gray, 40% transparent	#669F9F9F	Ripple, tether primary

3	Black, 40% transparent	#66000000	Touch glow outer gradient, tether accent

The default color values each have transparency and vary in intensity from white to black. If you change the defaults, it is important to have some variety so that at least one color has contrast against the various foreground and background colors in the application. Figure 10.2 shows three customized touch visualizations to highlight the role of the colors.

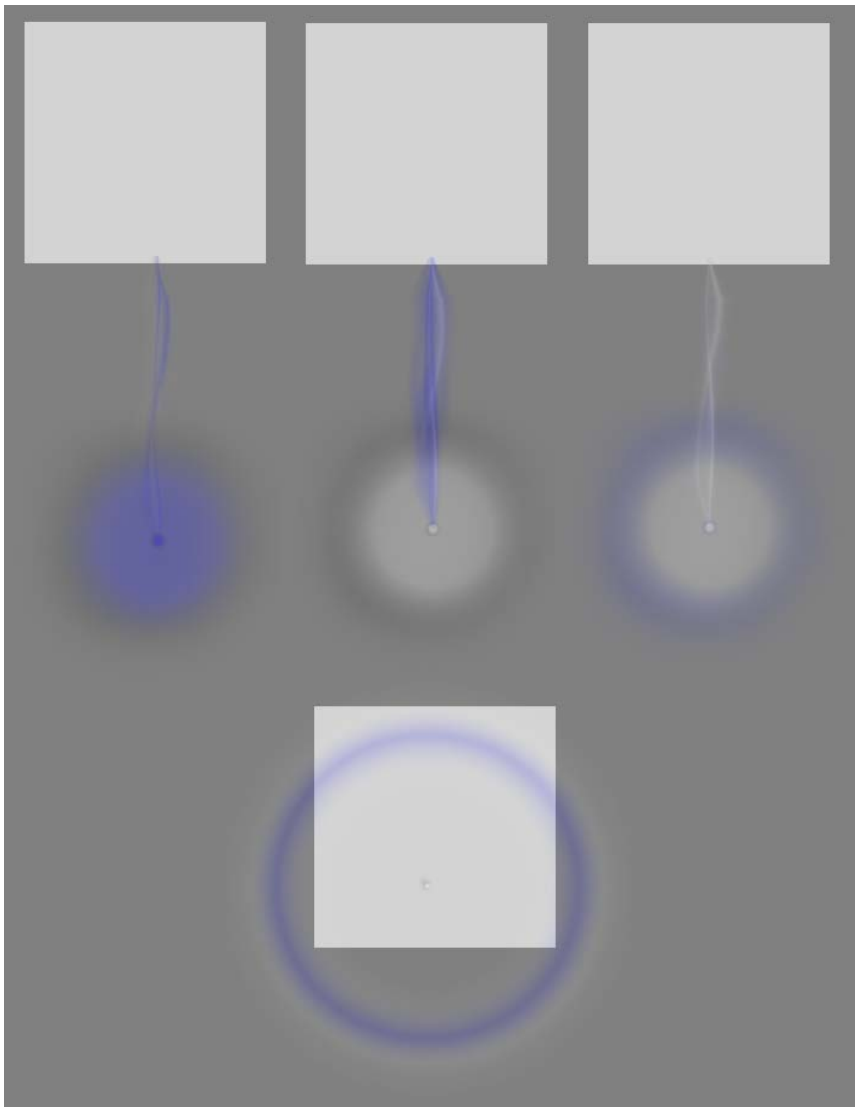


Figure 10.2 Three configurations of touch visualization colors. On the left, color 1 was set to blue. In the middle top and bottom, color 2 was set to blue. On the right, color 3 was set to blue. In all cases the other colors were left to default. The bottom middle shows a frame of the animated ripple effect as the TouchDevice is captured to the UIElement.

There are two ways to change the colors. First, you can set attach properties in XAML for each of the three colors. For example, this XAML will cause any touches that start within the Border to uses these colors:

```
<Border s:TouchVisualizer.VisualizationColor1="#66FFFFFF"
        s:TouchVisualizer.VisualizationColor2="#669F9F9F"
        s:TouchVisualizer.VisualizationColor3="#66000000">
```

These are the default colors, but you could also use other hex values or named colors.

NOTE

The XAML examples assume that the XML namespace "s" is defined in the root of the XAML document as the standard Surface namespace: xmlns:s="http://schemas.microsoft.com/surface/2008"

You can also change the colors in code:

```
UIElement element = GetUIElement();
TouchVisualizer.SetVisualizationColor1(element, color1);
TouchVisualizer.SetVisualizationColor2(element, color2);
TouchVisualizer.SetVisualizationColor3(element, color3);
```

There are also TouchVisualizer.GetVisualizationColor* methods that you can use to read the colors.

So far we have seen how to set attached properties to configure the visualization colors and whether to show visualizations. For greater control of when to show visualizations and the positions of tether anchors, we can also use TouchVisualizerAdapters.

CONTROLLING TETHERS WITH ADAPTERS

Tethers are displayed when a TouchDevice has been captured to an element and the position of the TouchDevice has left the boundary of the element. The tether is drawn between the position of the TouchDevice and an anchor point on the element. For this to work, the touch visualization framework needs to know the element boundaries and have a method to calculate anchor positions. We can tell the touch visualization framework this information using TouchVisualizerAdapters.

When you can set a TouchVisualizerAdapter on a UIElement, you're telling the Touch Visualization Framework how to behave when a TouchDevice is captured to a UIElement. The TouchVisualizerAdapter determines, for each captured TouchDevice, whether to show visualizations and the anchor location of the optional tether.

You can set a TouchVisualizerAdapter in XAML using this syntax:

```
<Border>
    <s:TouchVisualizer.Adapter>
        <s:TouchVisualizerAdapter />
    </s:TouchVisualizer.Adapter>
</Border>
```

Or in code:

```
UIElement element = GetElement();
TouchVisualizer.SetAdapter(element, new TouchVisualizerAdapter());
```

If you do not set a TouchVisualizerAdapter, then visualizations will be visible but no tethers will be drawn. You only need to set adapters on UIElements that are capturing TouchDevices, such as those with IsManipulationEnabled set to true.

The TouchVisualizerAdapter logic is only invoked when an InputDevice is captured to a UIElement, such as when IsManipulationEnabled is set to true, and only when visualizations are enabled. The ShowsVisualizations attached property overrides the TouchVisualizerAdapter logic.

Surface SDK provides several TouchVisualizerAdapters out of the box. In most cases, you can use one of these existing adapters, listed in Table 10.3.

Table 10.3 Surface SDK provides several TouchVisualizerAdapters out of the box

Adapter	Description
CompoundTouchVisualizerAdapter	Used in custom adapters to combine functionality with another adapter
TouchVisualizerRectangleAdapter	Provides a simple rounded rectangle behavior for tethers
SurfaceButtonTouchVisualizerAdapter	Provides the tether behavior used with SurfaceButtons
SurfaceScrollViewerTouchVisualizerAdapter	Provides the tether behavior used with SurfaceScrollViewers
TagVisualizerTouchVisualizerAdapter	Coordinates visualizations with a TagVisualizer
TouchVisualizerAdapter	Base class for adapters. Provides general purpose visualization logic for a variety of shapes

In most cases, the TouchVisualizerAdapter is sufficient. It has logic for various shapes and geometries. You could use TouchVisualizerRectangleAdapter instead, which does fewer calculations but only handles rectangle and rounded rectangle geometry.

The SurfaceButtonTouchVisualizerAdapter, SurfaceScrollViewerTouchVisualizerAdapter, and TagVisualizerTouchVisualizerAdapter each enable tether behaviors that make sense within the context of those controls. For example, the SurfaceButtonTouchVisualizerAdapter only shows tethers when two or more TouchDevices are captured, and animates the tether reaching out to the TouchDevice when it displays.

If the existing TouchVisualizerAdapters do not fit your needs, then you can derive your own with custom behaviors for use with your custom controls.

CUSTOM VISUALIZER ADAPTERS

While it is recommended that you take advantage of existing TouchVisualizerAdapters, you may have a control metaphor that could be enhanced by a custom adapter. To create a custom adapter, you'll need to derive from an existing adapter and then add your visualizer logic. The logic can be added in two overrides:

```
protected override bool ShowsVisualization(InputDevice inputDevice)
{ }
```

and

```
protected override TouchVisualizationAnchor GetAnchorPosition(
                                              [CA]InputDevice inputDevice)
{ }
```

The ShowsVisualization() override should return true if the specified InputDevice should show visualizations. The GetAnchorPosition() override should return a TouchVisualizationAnchor instance, which requires holds a Point for the position of the anchor. This Point should be relative to the IInputElement the InputDevice is captured to.

You only need to override the method that you need to change the behavior for. In both method, you must write logic based upon the provided InputDevice. In implementation, you might check properties the IInputElement that captured the device and you could also add dependency properties to the TouchVisualizerAdapter to allow configuration of behavior. Also keep in mind that you can call the base class implementation if you want to use it in some cases.

To illustrate the most basic custom TouchVisualizerAdapter, I have written a PointTouchVisualizerAdapter in the CustomTouchVisualizerAdapter sample project. This class defines a dependency property Position, which allows you to directly specify the anchor position of tethers. This might allow you to attach a tether directly to specific visual affordance on the UIElement. The following code shows the use of PointTouchVisualizerAdapter:

```
<Border IsManipulationEnabled="True">
    <s:TouchVisualizer.Adapter>
        <local:PointTouchVisualizerAdapter Position="0,100" />
    </s:TouchVisualizer.Adapter>
    <!-- Other visuals -->
</Border>
```

This code forces all tethers to be anchored to a specific position on the Border, as shown in figure 10.3.

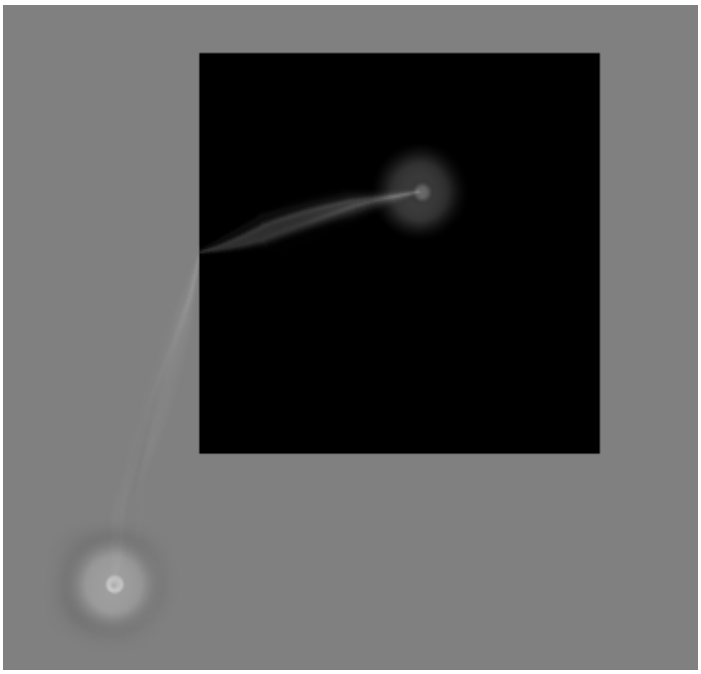


Figure 10.3 The CustomTouchVisualizerAdapter sample project shows how the PointTouchVisualizerAdapter forces all tethers to be attaches to a specific position relative to the element. In this cases, all tethers go to the (0,100) position within this 200 by 200 square.

The PointTouchVisualizerAdapter implementation is show in Listing 10.1.

Listing 10.1 PointTouchVisualizerAdapter attaches tethers to the specified point

```
public class PointTouchVisualizerAdapter : CompoundTouchVisualizerAdapter
{
    public Point? Position                                  #A
    {
        get { return (Point?)GetValue(PositionProperty); }
        set { SetValue(PositionProperty, value); }
    }

    public static readonly DependencyProperty PositionProperty =
        DependencyProperty.Register("Position", typeof(Point?),
        typeof(PointTouchVisualizerAdapter),
        new UIPropertyMetadata(null));

    protected override TouchVisualizationAnchor
        [CA]GetAnchorPosition(InputDevice inputDevice)
    {
        if (Position.HasValue)                              #B
        {
            return new TouchVisualizationAnchor(Position.Value);
        }
        return base.GetAnchorPosition(inputDevice);     #C
    }
}
```

#A Define a configurable Position DependencyProperty
#B Use Position if it is set
#C Otherwise use base class behavior

PointTouchVisualizerAdapter could be extended to support multiple configurable anchor positions and use the one closest to the InputDevice. Another extension might be to define an attached property for anchor affordance positions and search the element that captured the InputDevice for those properties.

In this section, we learned about the Touch Visualization Framework, the different parts of the touch visuals, and how to customize the visualizer behavior using existing or custom TouchVisualizerAdapters.

In the next section, we're going to learn about the other major framework provided by Surface SDK: the Drag-and-Drop framework.

10.2 Using Drag-and-Drop

Dragging and dropping objects is a common direct interaction, especially with touch interfaces. Drag-and-drop has been around since the beginning of GUI, but mostly as a simpler, icon-based interaction. We have opportunities for much richer interactions with multi-touch and so we need a better drag-and-drop system, such as the Drag-and-Drop Framework provided by Surface SDK.

This section will introduce the Surface SDK Drag-and-Drop Framework, show you the process for dragging content between controls, explore how to add drag-and-drop to custom controls, and wrap up with showing how to create smooth transitions between the drag phase and the drop phase.

10.2.1 Overview of the Drag-and-Drop Framework

The Surface Drag-and-Drop Framework enables your application's users to smoothly move content from one container to another container. It differs from the native WPF drag-and-drop in several ways. First, Surface Drag-and-Drop supports multi-touch and multi-users, which means multiple independent drags can occur at the same time. Second, Surface Drag-and-Drop allows rich dragging interactions with fully interactive data-bound visual. Finally, Surface Drag-and-Drop is designed solely for dragging within a single window. Many, if not most, Surface and Windows 7 touch applications will run full screen. If you wanted to support dropping files from Windows Explorer, then you would want to also support the native drag-and-drop.

You may not need drag-and-drop in every application, but anytime you want your users to move content from one container to another it would be a good idea to use the Surface Drag-and-Drop Framework. While there are other interactions possible, Surface Drag-and-Drop does a lot of the hard work for you and in almost all cases it is better to start with this existing framework. This section will get you up to speed with the Surface Drag-and-Drop concepts, classes, and events.

DRAGGING CONCEPTS

When you drag and drop an object, you're dealing with three concepts: the source, the cursor, and the target. The source is container the object came from and could be a LibraryStack, a SurfaceListBox, a ScatterView, or another control. The cursor represents the data in transit and has a visual that the user interacts with. The target is the UIElement that the cursor was dropped over.

The source and target elements are typically implemented as containers, consisting of an ItemsControl holding multiple pieces of content, although they do not need to be. They source or target could also be a ContentControl or other UIElement. This might be the case if you need an drag source that generates identical drag cursors every time or a drag target that is drop-only and performs an action such as transmit, save, or delete on the data that is dropped over it.

> **NOTE**
>
> Similar to WPF 4's raw touch events, Surface Drag-and-Drop events also work on UIElement, UIElement3D, and ContentElement. Again, for the sake of brevity, I will only mention UIElement when discussing the framework, but unless otherwise noted the same concepts and techniques also apply to UIElement3D and ContentElement.

The proposed target changes as the user drags the cursor around and is only finalized when the user releases all of the touches from the cursor. Targets much be configured to allow dropping by setting the UIElement.AllowDrop property to true, and they must be hit test visible. The framework identifies the proposed target by hit testing the center of the cursor visual against elements with AllowDrop set to true. You can also modify the hit test behavior subscribing to the SurfaceDragDrop.QueryTarget event.

Let's take a look at the classes in the framework and the events that are in the Surface Drag-and-Drop Framework.

DROPPING CLASSES AND EVENTS

The Surface Drag-and-Drop Framework consists of only two main classes and several supporting EventArgs classes, shown in table 10.4.

Table 10.4 The Surface Drag-and-Drop Framework classes

Class	Description
SurfaceDragDrop	Defines DragDrop events and utility methods
SurfaceDragCursor	Represents an object being dragged. Holds references to the dragged data, visual, and allows configuration of drag behavior
SurfaceDragDropEventArgs	Generic EventArgs used for most DragDrop events except the ones below
SurfaceDragCompletedEventArgs	Used for the DragCompleted event
GiveFeedbackEventArgs	Used for the GiveFeedback event
QueryTargetEventArgs	Used for the QueryTarget event
TargetChangedEventArgs	Used for the TargetChanged event

The SurfaceDragDrop class is the core of the framework as it defines the various events used for drag-and-drop and exposes several important methods for managing the drag operation. The SurfaceDragDrop methods are shown in table 10.5.

Table 10.5 The SurfaceDragDrop methods used for managing drag operations

Method	Description
BeginDragDrop	Creates a drag cursor and begins the drag operation
CancelDragDrop	Cancels a drag operation on the specified drag cursor
EndDragDrop	Cancels manipulation on the specific drag cursor. This is equivalent to the user lifting all fingers from the drag cursor.
GetAllCursors	Gets all active drag cursors within a window
GetCursorsFromDragSource	Gets all cursors that originated from the specified drag source
GetIsAnyCursorTargeted	Gets whether any cursors are targeting the specified element
GetTargetedCursors	Gets all cursors that are targeting the specified element

SurfaceDragCursor is the other primary class and represents the object being dragged. When BeginDragDrop is called, you provide it a FrameworkElement that the SurfaceDragCursor displays and the user manipulates. The SurfaceDragCursor instance lets you control the behavior of the cursor and the result of the drag operation.

During the drag operation, numerous events are raised that give you the opportunity to affect the behavior of the drag as well as provide visual feedback to the user. These attached events are defined on the SurfaceDragDrop class and are enumerated in table 10.6.

Table 10.6 SurfaceDragDrop events

Event	Preview event	Description
DragEnter	PreviewDragEnter	The cursor enters a UIElement boundary. Can set Effects.
DragLeave	PreviewDragLeave	The cursor left the UIElement boundary. Can set Effects.
DragOver	PreviewDragOver	The cursor is moving over a UIElement. Can set Effects.

GiveFeedback	PreviewGiveFeedback	Raised on the source element to allow it to provide feedback during the drag operation.
Drop	PreviewDrop	The cursor was dropped over a UIElement.
DragCanceled	PreviewDragCanceled	The drag operation was canceled programmatically.
DragCompleted	PreviewDragCompleted	The drag operation completed successfully.
QueryTarget	PreviewQueryTarget	Raised every drag cursor movement on each potential drop target to provide an opportunity to adjust the target
TargetChanged	PreviewTargetChanged	Raised when the proposed drop target has been changed by QueryTarget

These events can be set on any UIElement, UIElement3D, or ContentElement. You can use XAML syntax such as:

```
<Border AllowDrop="True"
        s:SurfaceDragDrop.Drop="Border_Drop" />
```

Or you can add the events in C#:

```
Border border = GetBorder();
border.AllowDrop = true;
SurfaceDragDrop.AddDropHandler(border,
         [CA]new EventHandler<SurfaceDragDropEventArgs>(Border_Drop));
```

Notice in both of these examples I am setting AllowDrop to true. This is a standard UIElement property and is the sample one used by the native drag and drop functionality. AllowDrop must be set to true for these events to be raised on an element. AllowDrop does not need to be set on a drag's source container if the source container does not need to be a drop target, but

Figure 10.4 illustrates a typical drag-and-drop sequence.

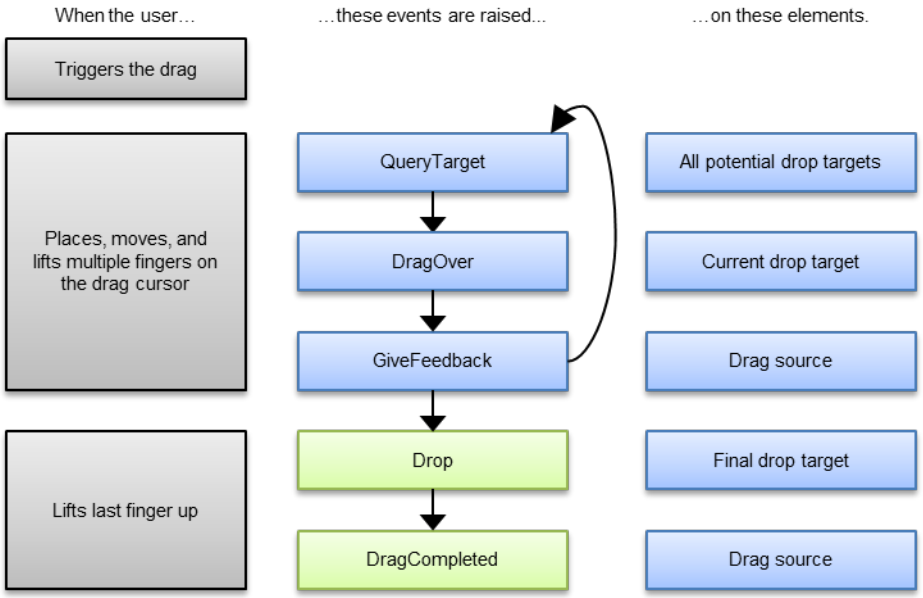


Figure 10.4 A typical drag-and-drop sequence. As the user drags the object from container to another, a variety of event

There are also the other events such as DragEnter and DragLeave that are triggered depending upon where the user drags or drops the cursor. The DragCanceled and TargetChanged events require code in order to be triggered. Let's learn more about these events by using them in practice.

10.2.2 Dragging between controls

The first thing to be aware of is that some of the Surface SDK controls already implement drag-and-drop. The LibraryStack, LibraryBar, and LibraryContainer already have all of the drag-and-drop logic integrated. Any time you use put those controls in an application, you can drag content from one to another without any extra code.

In many cases, you'll want to drag content from a library control to another control such as a ScatterView, a SurfaceListBox, or another custom control. In this section, we're going to learn how to enable drag-and-drop for any ItemsControl. We'll see how to start dragging, how to accept a drop, how to filter and cancel a drop, and how hit testing and layout affects drag-and-drop. In order to demonstrate these, we'll be using the SurfaceDragDropSample project, so let's take a look at the boilerplate of the sample first.

DRAG-AND-DROP SAMPLE

The SurfaceDragDropSample project is designed to illustrate a simple task: sorting content into categories by dragging from one container into another. I've chosen to sort fruits and vegetables in this sample, but the task could easily be adapted to sorting other items, such as products, packages, or people. Figure 10.5 shows the starting view of the application.

Figure 10.5 The SurfaceDragDropSample uses sorting of fruits and vegetables to demonstrate how to use the Surface Drag-and-Drop framework. The four rectangles with text are the content and the three labeled vertical areas are ItemsControls.

In order to easily demonstrate the operation of the Surface Drag-and-Drop Framework, I'm going to start out in the SurfaceDragDrop.1.Simple project. Later in this section I will show a refactored version that reflects a more real-world usage.

The sample includes simple FoodModel class, which contains Name and Type properties, and a basic FoodView UserControl to display those properties. The real guts of the sample are in the SurfaceWindow1.The interface contains three plain ItemsControls which are used as containers for the different roles within our mock sorting task: Fruits Only, Mixed, and Vegetables Only.

I'm using a plain ItemsControl here to both keep the sample simple as well as show that if you want to create a custom control, anything you see in this ItemsControl sample will apply. You may need to add

some custom behavior depending upon the specific control. For example, with a scrolling control such as SurfaceListBox, you need to figure out when the user intends to scroll and when the user intends to start a drag operation. For now, we will keep things simple. Listing 10.2 shows one of the ItemsControls.

Listing 10.2 An ItemsControls from the SurfaceDragDropSample

```
<ItemsControl Background="DarkGray"
              AllowDrop="True"
              PreviewTouchDown="items_PreviewTouchDown"
              s:SurfaceDragDrop.DragEnter="items_DragEnter"
              s:SurfaceDragDrop.DragLeave="items_DragLeave"
              s:SurfaceDragDrop.Drop="items_Drop"
              s:SurfaceDragDrop.DragCanceled="items_DragCanceled"
              s:SurfaceDragDrop.DragCompleted="items_DragCompleted"
              ItemsSource="{Binding FruitItems}"
              ItemTemplate="{StaticResource FoodTemplate}"
              Grid.Row="1"
              Grid.Column="0" />
```

The three ItemsControls are each the same except for the grid positions and the ItemsSources. The ItemTemplate is set to a DataTemplate resource named FoodTemplate:

```
<DataTemplate x:Key="FoodTemplate">
    <local:FoodView />
</DataTemplate>
```

Each ItemsControl is also subscribed to several of the SurfaceDragDrop events. In the next few sections, I'll explain what each of the SurfaceDragDrop event handlers do and how they work.

STARTING THE DRAG

The drag operation is started in code and you need to figure out when exactly the drag should start. In this example, we want to start the drag operation whenever the user touches one of our TouchViews. We have subscribed to the PreviewTouchDown event and the event handler, showing in listing 10.3, contains the logic for starting the drag. We use the preview event here in case something within the ItemsControl or the ItemsControl itself handled the TouchDown event. This is the case with a SurfaceScrollViewer or SurfaceListBox where a manipulation captures the touches and controls scrolling.

Listing 10.3 The PreviewTouchDown event handler starts the drag operation

```
Dictionary<SurfaceDragCursor, UIElement> CursorToSourceVisual =
    new Dictionary<SurfaceDragCursor, UIElement>();

private void items_PreviewTouchDown(object sender, TouchEventArgs e)
{
    FrameworkElement source = sender as FrameworkElement;
    var element = e.TouchDevice.DirectlyOver as FrameworkElement;

    var view = VisualUtility.FindVisualParent<FoodView>(element);
    if (view == null || view.DataContext == null)
        return;                                                  #1

    var data = view.DataContext;

    List<InputDevice> devices = new List<InputDevice>();
    devices.Add(e.TouchDevice);

    var cursorView = new FoodView();
    cursorView.DataContext = data;

    SurfaceDragCursor cursor =                          #2
        SurfaceDragDrop.BeginDragDrop(source,           #2
                                      view,             #2
                                      cursorView,       #2
                                      data,             #2
                                      devices,          #2
                                      DragDropEffects.Move);

    view.Visibility = System.Windows.Visibility.Hidden;    #3
    CursorToSourceVisual.Add(cursor, view);                #3
```

```
}
  #1 Can't find FoodView in visual tree
  #2 Start the drag operation
  #3 Hide source view, associate with cursor
```

Cueballs in code and paragraphs below

This event handler finds the FrameworkElement that the touch is directly over. This could be an element within our FoodView control such as a TextBox or Border, or it could be part of the ItemsControl such as the background. We use VisualUtility.FindVisualParent<>(), which is a helper method from Blake.NUI, to search up the visual tree for a type of FoodView.

If we cannot find one, it might mean we touched something else within the ItemsControl template that was not a FoodView. In this case, the view variable will be null and we return since we cannot start a drag #1. We also return if the found FoodView does not have a DataContext for some reason. This is just a sanity check and wouldn't occur in this sample.

Once we have the view, we prepare to begin the drag operation. In order to start the drag, we call SurfaceDragDrop.BeginDragDrop() #2. Table 10.7 explains the parameters of the BeginDragDrop method.

While the user is manipulating the cursor visual around, the data is still in the ItemsControl's collection. Since we want to move the data, it makes sense that we set the visibility of the source visual to hidden and associate the cursor to the source visual in the CursorToSourceVisual dictionary #3. This will hide the source visual but preserve the empty space within the drag source ItemsControl. We will use CursorToSourceVisual in the DragCanceled event handler so that if a drop is cancelled so we can show the original visual again.

Table 10.7 The BeginDragDrop method requires these method call parameters

Type	Parameter	Description
FrameworkElement	dragSource	The container the data is being dragged from
FrameworkElement	draggedElement	The source visual element for the drag. The framework will copy the position, orientation, and size from this element to the cursorVisual
FrameworkElement	cursorVisual	A new instance of a visual that will be displayed during the drag operation. If binding is used, you should set the DataContext to the data.
object	Data	The actual data that is being transferred. This is typically a business object or ViewModel and was the DataContext of the draggedElement.
IEnumerable<InputDevice>	inputDevices	A collection of the inputDevices that should be automatically captured by the drag cursor visual.
DragDropEffects	allowedEffects	The effects desired by the drag source.

Once you have started the drag, the user can manipulate the cursor visual around the window. Behind the scenes, the cursor visual is actually added to a ScatterView that the Drag-and-Drop framework creates within an adorner layer. This means that the interaction with a cursor is the same as a ScatterViewItem. You can also configure whether the cursor can scale or rotate by setting the SurfaceDragCursor.CanScale or SurfaceDragCursor.CanRotate properties.

If you only added code to call BeginDragDrop and did nothing else, then you would be able to drag content and drop it but nothing would happen on the drop. The dragged data would still be in the drag source and no data would be moved. In order to actually take action upon a drop, you'll need to add more event handlers.

ACCEPTING THE DROP

After starting the drag, the primary event you need to handle is SurfaceDragDrop.Drop. This event is fired when a SurfaceDragCursor is dropped over a UIElement that has AllowDrop set to true. Within this event, you need to do something with the dropped cursor and data. Listing 10.4 shows the drop event handlers used in the SurfaceDragDropHandler.

Listing 10.4 The drop event handler adds the data to the target container

```
private void items_Drop(object sender, SurfaceDragDropEventArgs e)
{
    ItemsControl target = sender as ItemsControl;               #1
    var targetCollection = target.ItemsSource                   #1
                           [CA]as ICollection<FoodModel>;       #1

    var data = e.Cursor.Data as FoodModel;
    if (data != null)
        targetCollection.Add(data);                             #2
    e.Handled = true;
}
```

#1 Get reference to target collection
#2 Add cursor data to target

Cueballs in code and text

In this code, we look at the sender of the event to find the ItemsControl the drop event was fired upon and get the collection that backs the ItemsControl #1. Next, we can add the cursor's data to that collection #2, which will append it to the end of the ItemsControl.

> **NOTE**
>
> It is important to set e.Handled to true for all of the Drag-and-Drop events. If you don't, the event will bubble up and could result in hard-to-diagnose bugs if handled higher in the visual tree.

After the drop the data is still in the source collection, so if we did nothing else, then this would result in a copy operation. In this sample, we want to move content rather than copy it. In order to make that happen, the drag source needs to take some action during the DragCompleted event. We will see how that works shortly, but first, let's take a look at how the DragDropEffects can be used to change whether a drop will result in a copy, move, or no action.

MODIFYING THE DROP EFFECTS

The BeginDragDrop method call specifies default AllowEffects, which of type DragDropEffects, enumerated in table 10.8. The AllowedEffects value is copied to the SurfaceDragCursor.AllowedEffects property and does not change.

Table 10.8 The System.Windows.DragDropEffects enumeration and MSDN documentation

Enum	Description
All	The data is copied, removed from the drag source, and scrolled in the drop target
Copy	The data is copied to the drop target
Link	The data from the drag source is linked to the drop target
Move	The data from the drag source is moved to the drop target
None	The drop target does not accept the data
Scroll	Scrolling is about to start or is currently occurring in the drop target

These descriptions are from the MSDN documentation and unfortunately are vague. They were written for drag-and-drop between windows and the original meanings may not apply for all situations. For

example, Link might be used when dragging a file into a document to embed it and link to the original, but your application may not need to do that. In most cases you will only need to deal with the Move, Copy, or None values.

Event handlers such as DragEnter, DragOver, and DragLeave can change the desired Effect by setting SurfaceDragDropEventArgs.Effects. Listing 10.5 shows how the SurfaceDragDropExample project handles DragEnter and DragLeave.

Listing 10.5 The DragEnter and DragLeave events filter potential drop targets

```
void items_DragEnter(object sender, SurfaceDragDropEventArgs e)
{
    ItemsControl target = sender as ItemsControl;
    var data = e.Cursor.Data as FoodModel;

    if (data == null ||                                          #1
        e.Cursor.DragSource == target ||                         #1
        (data.Type == FoodType.Fruit &&                          #1
         target.ItemsSource == VegetableItems) ||                #1
        (data.Type == FoodType.Vegetable &&                      #1
         target.ItemsSource == FruitItems))                      #1
    {
        e.Effects = DragDropEffects.None;                        #2
    }
    e.Handled = true;
}

void items_DragLeave(object sender, SurfaceDragDropEventArgs e)
{
    e.Effects = e.Cursor.AllowedEffects;                         #3
     e.Handled = true;
}
```

#1 If incompatible drop target
#2 Then don't allow the drop
#3 Reset to default values

Cueballs in code and text

This code checks to see if the drop target is compatible with the cursor data when the cursor is dragged over a drop target. In this sample, we simply check the FoodModel's type against the container which raised the DragEnter event #1. If the drop would put a Fruit into the Vegetable container or vice versa, or drop the cursor into the drag source, then we set the e.Effects to None #2. If the user drops the cursor when the effects have been set to None, then the DragCanceled event is raised instead of the default of Drop and DragCompleted.

> **NOTE**
>
> Internally, the SurfaceDragDropEventArgs.Effects property is copied to the Cursor.Effects property after the DragEnter, DragOver, or DragLeave event is called. Cursor.AllowEffects and Cursor.Effects are read-only, while SurfaceDragDropEventArgs.Effects is read-write.

If the user drags the cursor out of a drop target, then the drop target resets the Effects to the original effects specified in the BeginDragDrop, accessible via the e.Cursor.AllowedEffects property #3. In this case, AllowedEffects would be Move, as this was the value we used when we called BeginDragDrop in listing 10.3.

In this sample, we're only using DragEnter and DragLeave to change the DragDropEffects. In a real application, we would want to also give visual feedback to the user about what will happen when the user drops the cursor. We will see an example of this in section 10.2.3. For now, let's continue and see what we need to do on the drag source side to clean up after the cursor is dropped.

CLEANING UP AFTER THE DROP

When the user drops the cursor, there are two possible results. Typically, the Drop event is raised on the drop target and then the DragCompleted event is raised on the drag source. On the other hand, the DragCanceled event might be is raised on the drag source in place of Drop and DragCompleted. This would occur if the cursor is not dropped over a drop target, an event handler set the Effects are set to None, or the drag operation is programmatically cancelled by calling:

```
SurfaceDragDrop.CancelDragDrop(cursor);
```

The DragCompleted and DragCanceled events give the drag source a chance to clean up after the drag operation is completed. This cleanup might include removing the data from the source collection if the DragDropEffects are set to Move or updating the source visual. Let's take a look at how the SurfaceDragDropSample sample handles these two events. Listing 10.6 shows the DragCompleted event handlers.

Listing 10.6 The DragCompleted event handler removes data from the source collection

```
private void items_DragCompleted(object sender,
                SurfaceDragCompletedEventArgs e)
{
    ItemsControl source = sender as ItemsControl;
    var data = e.Cursor.Data as FoodModel;
    if ((e.Cursor.Effects & DragDropEffects.Move) ==                    #1
        [CA]DragDropEffects.Move)                                       #1
    {
        var sourceCollection = source.ItemsSource
                            [CA]as ICollection<FoodModel>;
        if (sourceCollection.Contains(data))
        {
            sourceCollection.Remove(data);                              #2
        }
    }

    if (CursorToSourceVisual.ContainsKey(e.Cursor))                     #3
    {                                                                   #3
        CursorToSourceVisual.Remove(e.Cursor);                          #3
    }                                                                   #3
    e.Handled = true;
}
```

#1 If Effects include Move
#2 Then remove data from source collection
#3 Remove cursor reference in local dictionary

Cueballs in code and text

This DragCompleted event handler checks whether the Effects resulted in a Move #1 and if so, remove the cursor data from the source collection #2. It also removes the cursor from a dictionary we're using to associate the cursor with the source visual #3. Notice that when we check the effects, we check e.Cursor.Effects rather than just e.Effects like in the DragEnter method.

If you recall the PreviewTouchDown event handler from listing 10.3, you'll remember that we set the Visibility of the source visual to Hidden. This preserves the space in the source list but gives the impression that we have moved content as the cursor. The CursorToSourceVisual associates the cursor to the source visual so that we can reset the visibility if the drag operation is cancelled. We take care of this in the DragCanceled event handler, shown in listing 10.7.

Listing 10.7 The DragCanceled event handler resets the visuals to how they were

```
void items_DragCanceled(object sender, SurfaceDragDropEventArgs e)
{
    if (CursorToSourceVisual.ContainsKey(e.Cursor))
    {
        CursorToSourceVisual[e.Cursor].Visibility =            #1
            [CA]System.Windows.Visibility.Visible;             #1
        CursorToSourceVisual.Remove(e.Cursor);
    }
```

```
    e.Handled = true;
}
```

#1 Reset the source visual to Visible

Cueballs in code and text

If the DragCanceled event is called then we need to reset things the way they were before the drag started. In this sample we are setting the source visual back to Visible and removing the cursor to source visual association.

So far the sample has only used plain ItemsControls. We can also use controls that derive from ItemsControl in this sample. Depending upon the type of control, you may need to take into account the behavior of the control so the user easily understands how to use it and how to drag content.

10.2.2 Integrating with specific controls

In our sample, we have been doing all of the drag and drop code in the window's code behind. This was good for learning purposes but is pretty messy. Let's do some refactoring to see what a more production-quality architecture might look like. While we're at it, we will also replace the ItemsControls with other controls that derive from ItemsControl such as LibraryStack, SurfaceListBox, and ScatterView. The sample code that includes the modifications for this section is in the SurfaceDragDrop.2.Refactored folder. Figure 10.6 shows the refactored interface.

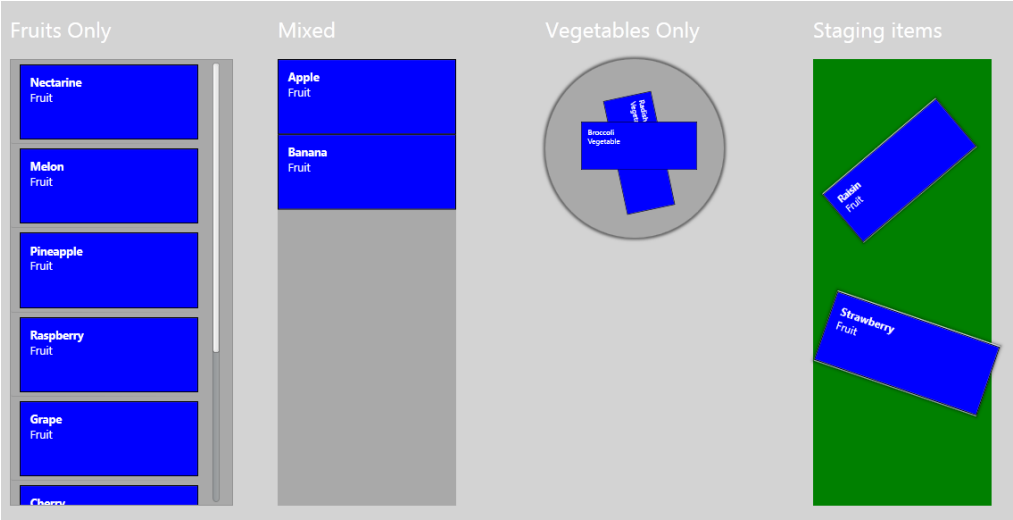


Figure 10.6 The SurfaceDragDrop.2.Refactored sample replaces the bare ItemsControls with more practical controls deriving from ItemsControl. From left to right, the controls shown are a DragDropSurfaceListBox, a CustomItemsControl, a LibraryStack, and a DragDropScatterView. Content can be dragged between any of these controls, as long as you don't put fruits in the vegetable stack or vegetables in the fruit list!

The process of refactoring the project was pretty straight-forward. In general, we replace the ItemsControl with other controls with reusable drag-and-drop behaviors. We keep the DragEnter and DragLeave events at the SurfaceWindow though because those events do application-specific filtering of content and are not reusable.

Let's go through each control refactoring.

ADDING A LIBRARYSTACK

Starting with the easiest replacement, let's replace the "vegetables only" ItemsControl on the right with a LibraryStack. The new XAML is shown in listing 10.8.

Listing 10.8 The LibraryStack as used in the SurfaceDragDrop sample

```
<Canvas Grid.Row="1"                                          #1
        Grid.Column="4">
    <s:LibraryStack
        Background="DarkGray"
        Height="200"
        Width="200"
        s:SurfaceDragDrop.DragEnter="items_DragEnter"          #2
        s:SurfaceDragDrop.DragLeave="items_DragLeave"          #2
        ItemsSource="{Binding VegetableItems}"
        ItemTemplate="{StaticResource FoodTemplate}" />
</Canvas>
```

#1 Use Canvas to fix clipping issues
#2 Keep the DragEnter and DragLeave events

Cueballs in code and text

The LibraryStack has drag-and-drop support built-in, so all we had to do was replace the control name ItemsControl with s:LibraryStack and remove all of the drag-and-drop events and properties except DragEnter and DragLeave #2.

Recall that the LibraryStack has animated transitions when you flip through the stack and when a drag is canceled. Since we're embedding the LibraryStack within a Grid, those animations may be clipped at the grid cell boundaries. To work around that, we can simply wrap the LibraryStack in a Canvas #1, which eliminates the boundary clipping problem. This problem does not occur when the LibraryStack is within a ScatterView.

The LibraryStack does a lot of the work for us. We can also create custom, reusable controls that package up drag-and-drop behaviors. Let's refactor our own event handlers into a custom control.

BUILDING A CUSTOM CONTROL

In the current sample project, I've created a custom control called CustomItemsControl. All I did it take our previously written drag-and-drop event handlers and moved them into the control's code. Listing 10.9 shows how to use the CustomItemsControl.

Listing 10.9 CustomItemsControl

```
<controls:CustomItemsControl
    Background="DarkGray"
    s:SurfaceDragDrop.DragEnter="items_DragEnter"
    s:SurfaceDragDrop.DragLeave="items_DragLeave"
    ItemsSource="{Binding MixedItems}"
    ItemTemplate="{StaticResource FoodTemplate}"
    Grid.Row="1"
    Grid.Column="2" />
```

Just like the LibraryStack, we keep the DragEnter and DragLeave events on this level but the rest of the reusable drag-and-drop behavior is moved into the control.

Organizing custom control resource dictionaries

When I created the CustomItemsControl, in Visual Studio I created a Controls folder, right clicked, and used the Add New Item command. In the dialog, I selected "Custom Control (WPF)" as the template and gave it the name CustomItemsControl.cs. It generated this file in the Controls folder and also added a file Generic.xaml under the Themes folder. Generic.xaml is a resource dictionary that contains styles for controls.

As a matter of convention and better organization, I always move things around a little bit when I add custom controls. In this case, I did Add New Item again on the Controls folder and added a Resource Dictionary called CustomItemsControl.xaml. I then copied the style from Generic.xaml to this new resource dictionary. This allows me to easily find the style, similar to how User Controls group the XAML and code-behinds.

Since WPF looks to Themes/Generic.xaml for custom control styles, we need to do one more step to make sure it knows to look for our control style. We need to add a MergedDictionary collection in Generic.xaml and give it a pointer to our new resource dictionary. Generic.xaml should look something like:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="/SurfaceDragDropSample;
            [CA]component/Controls/CustomItemsControl.xaml" />
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

It is important to use the "/AssemblyName;component/Path/FileName.xaml" format for the Source property. This ensures that WPF looks in the correct assembly, even when this assembly is being referenced by another.

You can add additional ResourceDictionary lines in the MergedDictionaries section as you add more custom controls to your project.

If you're wondering what makes WPF look for a style, it is this line in the static constructor of a custom control:

```
DefaultStyleKeyProperty.OverrideMetadata(
    typeof(CustomItemsControl),
    new FrameworkPropertyMetadata(typeof(CustomItemsControl)));
```

The two previous code samples should be in sidebar

This line and the static constructor are automatically added with the custom control template and it is best to leave it there.

From a purely functional standpoint, the CustomItemsControl does not really need a control template defined in CustomItemsControl.xaml since we don't change the default style. I could have removed the style and the DefaultStyleKeyProperty line from the static constructor, but I left it in there in case you want to use this as a basis for a more advanced control.

In the CustomItemsControl.cs file, we need to set the same properties and events as in our previous code sample. I've added these lines to the constructor:

```
Background = Brushes.Transparent;
this.Loaded += new RoutedEventHandler(CustomItemsControl_Loaded);
this.Unloaded += new RoutedEventHandler(CustomItemsControl_Unloaded);
```

We set the Background to Transparent to make sure the control is visible in hit testing. During the loading process, WPF will overwrite this value if the user of the control has set another background color, such as DarkGray in our case. We should wait until the control is Loaded to add the drag-and-drop events. This avoids any rare race conditions that might cause one of these events to be called before the control is ready. Listing 10.10 shows the Loaded and Unloaded events.

Listing 10.10 Adding and removing drag-and-drop handlers

```
void CustomItemsControl_Loaded(object sender, RoutedEventArgs e)
{
    SurfaceDragDrop.AddDropHandler(this, OnCursorDrop);
    SurfaceDragDrop.AddDragCompletedHandler(this, OnDragCompleted);
    SurfaceDragDrop.AddDragCanceledHandler(this, OnDragCanceled);
    this.AllowDrop = true;
}

void CustomItemsControl_Unloaded(object sender, RoutedEventArgs e)
{
    this.AllowDrop = false;
    SurfaceDragDrop.RemoveDropHandler(this, OnCursorDrop);
    SurfaceDragDrop.RemoveDragCompletedHandler(this, OnDragCompleted);
```

```
        SurfaceDragDrop.RemoveDragCanceledHandler(this, OnDragCanceled);
    }
```

This code ensures the event handlers are added and removed symmetrically. This is important if the control were to be removed from the visual tree and added again later.

The OnCursorDrop, OnDragCompleted, and OnDragCanceled event handlers are almost the same as in the first SurfaceDragDropSample code so I won't list them here. The main difference is it adds and removes items from the ItemsSource if the ItemsControl is data bound to an ItemsSource, or otherwise adds and removes items directly from the Items collection. This adds flexibility to be used in both regular and data bound scenarios.

The final component we're missing is starting the drag. Previously, we subscribed to PreviewTouchDown for the entire ItemsControl then started from the element the touch was directly over, searched upwards in the visual tree for a FoodView element. Since we're trying to make this control reusable, we can't depend upon this same technique.

For this example, we subscribe to PreviewTouchDown for each individual ItemContainer. We can get access to the ItemContainer by overriding the PrepareContainerForItemOverride method and subscribing to the PreviewTouchDown on the container:

```
    protected override void PrepareContainerForItemOverride(
        [CA]DependencyObject element, object item)
    {
        FrameworkElement container = element as FrameworkElement;
        if (container != null)
            container.PreviewTouchDown += container_PreviewTouchDown;
        base.PrepareContainerForItemOverride(element, item);
    }
```

To keep things symmetrical, we also need to override ClearContainerForItemOverride and unsubscribe the PreviewTouchDown event handler. The container_PreviewTouchDown method is nearly the same as in the previous sample, so I will also skip listing it.

Let's move on to see how we added drag-and-drop to a real SurfaceListBox.

ADDING DRAG-AND-DROP TO SURFACELISTBOX

The standard SurfaceListBox does not support drag-and-drop, but it does not take too much effort to add it. I've included the reusable DragDropSurfaceListBox control in the Blake.NUI library. Check http://blakenui.codeplex.com for the latest code.

We do not need to change the control template so all we're doing is deriving from SurfaceListBox and adding some code. DragDropSurfaceListBox.cs includes all of the code from CustomItemsControl plus a little extra that makes drags behave well with scrolling.

In CustomItemsControl, we started the drag during PreviewTouchDown. For DragDropSurfaceListBox, we actually want to wait a little bit to see if the user is scrolling vertically or pulling an item out horizontally for dragging. We do this by tracking new touches and starting a drag if the movement is primarily horizontal or ignoring the touches if the movement is primarily vertical.

This ends up being a rare case where we want to track a manipulation independent of a UIElement, so it gives me a chance to use the low-level ManipulationProcessor2D class. This class is pure math and independent of any visualization, which makes it perfect for what we need to do here. When you have a chance, check the implementation of DragDropSurfaceListBox to see how this works in detail.

> **NOTE**
>
> There are also other techniques for starting a drag from within a SurfaceListBox or SurfaceScrollViewer. One idea might be to have a handle affordance on each element and only start the drag when the user touches the drag handle.

In DragDropSurfaceListBox, we subscribe to TouchDown, TouchMove, and TouchUp and use a ManipulationProcessor2D to track each TouchDevice. When a TouchDevice has moved horizontally past a threshold, we interpret this as the user wanting to start the drag and we call a method called StartDrag().

This method has similar code as our previous examples and ends up calling SurfaceDragDrop.BeginDragDrop().

Now that we have the DragDropSurfaceListBox integrated, I want to add drag-and-drop to a ScatterView.

INTEGRATING THE SCATTERVIEW

For illustrative purposes, I have kept the ScatterView confined to its own column for now. (Reference the green rectangle in figure 10.6.) I've given the ScatterView a similar treatment as the SurfaceListBox and have added a DragDropScatterView control to Blake.NUI.

> **NOTE**
>
> DragDropScatterView is a modified version of the same named class from the Surface SDK samples. It includes an attached property AllowDrag that lets you enable or disable drag operations on each ScatterViewItem.

Integrating drag-and-drop with a ScatterView is somewhat interesting because the ScatterView inherently already has drag-and-drop, at least within itself, and the Drag-and-Drop Framework itself also uses a ScatterView for interacting with drag cursors.

The resolution ends up being that the items within the ScatterView are only displayed when they are not being manipulated. Once a manipulation starts, we start the drag and remove the data from the ScatterView.

In our other controls, when a drag started we would hide the dragged item within the source container and then either remove it on drag completion or show it again on drag cancelation. Whenever a cursor is dropped over the ScatterView, though, we need to set the ScatterViewItem to have the same position, orientation, and size as the cursor to ensure a smooth transition. This might occur when dragging content into the ScatterView or when dropping content that came from the ScatterView.

It ends up that the simplest way to handle this is to remove the data from the ScatterView on drag start and then re-add it in the new position, orientation, and size if it is dropped over the ScatterView. DragDropScatterView implements this in two methods. The first is OnManipulationStarted, shown in listing 10.11, which is the event handler for the SVI ContainerManipulationStarted event.

Listing 10.11 ScatterView starts the drag when the SVI is manipulated

```
private void OnManipulationStarted(object sender, RoutedEventArgs args)
{
    ScatterViewItem svi = args.OriginalSource as ScatterViewItem;
    if (svi == null || !DragDropScatterView.GetAllowDrag(svi))
        return;

    object data = svi.DataContext;

    ContentControl cursorVisual = new ContentControl();
    cursorVisual.Content = data;
    cursorVisual.ContentTemplate = this.ItemTemplate;

    SurfaceDragCursor cursor =
        SurfaceDragDrop.BeginDragDrop(this, svi,              #A
            cursorVisual, data,                               #A
            svi.TouchesCaptured, DragDropEffects.Move);       #A
    if (cursor == null)
        return;
    cursor.CanScale = svi.CanScale;
    cursor.CanRotate = svi.CanRotate;

    IList collection = ItemsSource as IList;                    #B
    if (collection == null)                                     #B
    {                                                           #B
        collection = Items;                                     #B
    }                                                           #B
    collection.Remove(data);                                    #B
}
```

#A Start the drag
#B Remove item from ItemsSource/Items

When a drag cursor is dropped over the DragDropScatterView, regardless of source, we will need to add it and configure the location of the SVI. We also need to handle the case of a drag starting from the ScatterView but being canceled. We reuse the same event handler, shown in listing 10.12, for both drop and drag canceled since they require the same code. The one bit of code added for the drag canceled scenario is a call to ScatterViewItem.BringIntoBounds().

Listing 10.12 Dropping a drag cursor over the DragDropScatterView

```
private void OnCursorDrop(object sender, SurfaceDragDropEventArgs args)
{
    SurfaceDragCursor droppingCursor = args.Cursor;

    if (droppingCursor.CurrentTarget == null ||
        droppingCursor.CurrentTarget == this)
    {
        if (!Items.Contains(droppingCursor.Data))
        {
            IList collection = ItemsSource as IList;
            if (collection == null)
            {
                collection = Items;
            }
            collection.Add(droppingCursor.Data);                        #A
        }

        var svi =
            ItemContainerGenerator.ContainerFromItem(droppingCursor.Data)
                as ScatterViewItem;
        if (svi != null)
        {
            svi.CanScale = droppingCursor.CanScale;                     #B
            svi.CanRotate = droppingCursor.CanRotate;                   #B
            svi.Center = droppingCursor.GetPosition(this);              #B
            svi.Orientation = droppingCursor.GetOrientation(this);      #B
            svi.Height = droppingCursor.Visual.ActualHeight;            #B
            svi.Width = droppingCursor.Visual.ActualWidth;              #B
            svi.SetRelativeZIndex(RelativeScatterViewZIndex.Topmost);   #B

            DragDropScatterView.SetAllowDrag(svi, true);
            if (svi.IsLoaded)
            {
                svi.BringIntoBounds();                                  #C
            }
            else
            {
                svi.Loaded += (s, e) =>
                    {
                        svi.BringIntoBounds();              #C
                    };
            }
        }
    }
}
```

#A Add item to ItemsSource/Items
#B Make SVI match cursor
#C Bring into bounds if drag canceled

At this point, we have integrated four different types of ItemsControls into our sample and we can drag content between them. The sample project is also much cleaner and closer to production ready code.

I want to show one last scenario that helps illustrate how to get the ScatterView work better with drag-and-drop in your interface.

GETTING THE LAYOUT RIGHT

Right now the ScatterView is isolated and does not overlap any other controls. I did this to show that even though it has its own moving layout, it acts just like any other ItemsControl. In many interfaces, you'll want to have a ScatterView behind other controls but still be able to drag content into them.

In order to do this, we need to mess with the layout a little bit. I have a DragDropScatterView commented out in the beginning of SurfaceWindow1.xaml before the Grid with rows and columns. Uncomment that ScatterView and comment out the other ScatterView within the Grid.

Now if you run the application, you'll see the green ScatterView is covering the entire background of the window. You can drag content out over the green background and drop it, move it around, and then drag it into another container.

The only problem now is that if you drop, for example, a fruit item over the vegetable only LibraryStack, it will refuse the item and the item will fall back behind the LibraryStack. This behavior is correct, but confusing. There are also other scenarios where you can get the item in the ScatterView to be behind the other containers when it seems like it should not be.

We could move the ScatterView to the front of the other visual elements. This presents a dilemma though, because then if the ScatterView is hit test visible, it would block the other containers from getting drop events.

There are probably several solutions for this problem, but here is one that worked well in this situation. I added a DependencyProperty to DragDropScatterView that allows me to specify an external UIElement to serve as the drop target. This let me put a (green again) border behind everything to serve as the drop target but keep the DragDropScatterView itself in front of everything with a null background.

NOTE

Elements with any color value including "Transparent" are visible to hit testing. Setting a color to null (x:Null in XAML) looks the same as transparent but is invisible to hit testing.

This solution can be found in the SurfaceDragDrop.3.Final folder in the sample code. Listing 10.13 shows an abbreviated XAML of the final sample's control layout.

Listing 10.13 Abbreviated XAML of final control layout

```
<s:SurfaceWindow>
    <Grid>
        <Border Background="Green"                                  #A
                Name="dropTarget" />                                #A
        <Grid>
            <b:DragDropSurfaceListBox />
            <c:CustomItemsControl />
            <Canvas>
                <s:LibraryStack />
            </Canvas>
        </Grid>
        <b:DragDropScatterView                                      #B
            Background="{x:Null}"                                   #B
            DropTarget="{Binding ElementName=dropTarget}" />        #B
    </Grid>
</s:SurfaceWindow>
```

#A External drop target for ScatterView
#B ScatterView with invisible background

You can see in this XAML that while the DragDropScatterView is in front, it has a null background and references a border in the back for catching dropped drag cursors. The allows ScatterViewItems to overlap in from of the containers but still be droppable in them. Figure 10.7 shows a visualization of the window.

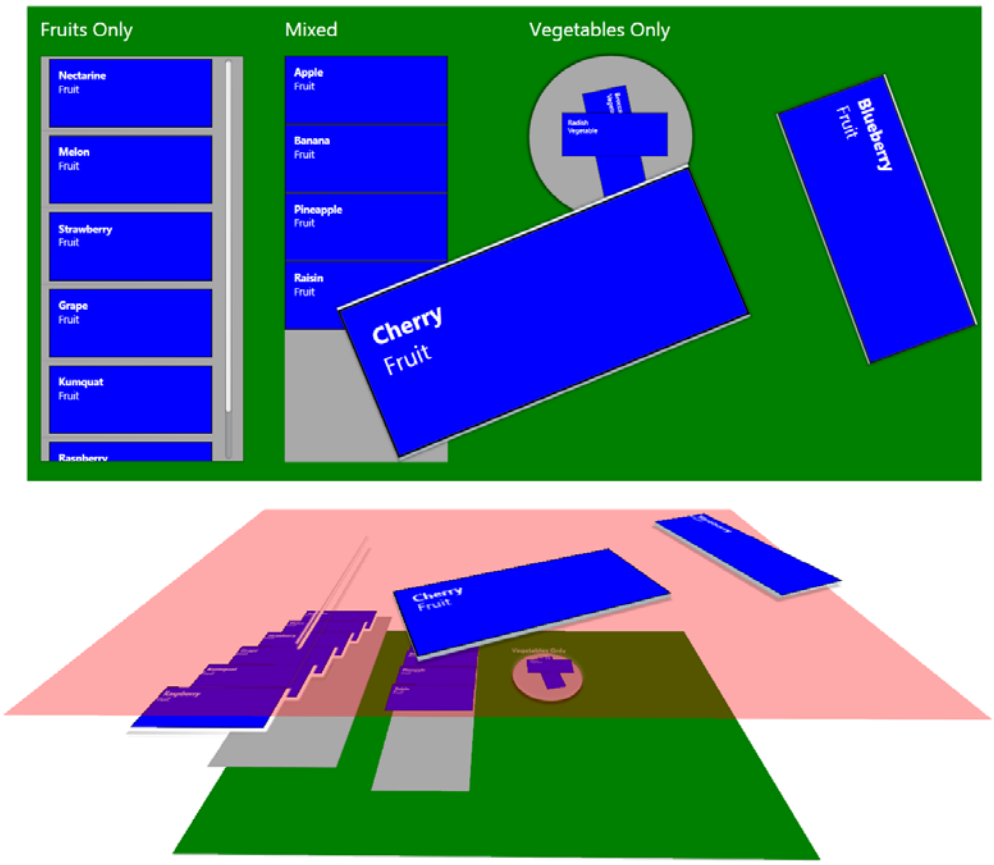


Figure 10.7 The top shows the final layout of the visual elements in 2D. The bottom visualizes the z-order of the visual elements in 3D using the Snoop tool. In the 3D view, a light tint was added to the DragDropScatterView for illustrative purposes, even though it has a null background. When dropping a drag cursor, the drop hit testing passes through the DragDropScatterView and could hit any of the containers. If it doesn't, the green background, serving as the DragDropScatterView's external drop target, is hit, and the item is added to the DragDropScatterView in the front.

We have now completed the sample illustrating the Surface Drag-and-Drop Framework. This sample could be extended further by putting the LibraryStack and other containers within the ScatterView. The DragDropListBox could also be enhanced by adding animated transitions for the dropping animation similar to the Library controls.

In this section we learned about the Drag-and-Drop Framework, how to use each of the events, and how to integrate the events with ItemsControls. We also saw how more practical controls were just extensions of the ItemsControls ideas and typically not much more complicated. We finally saw how to integrate drag-and-drop with a ScatterView and how to organize the layout to achieve the desired effects.

10.3 Summary

In this chapter, we have learned about two frameworks provided by the Surface SDK: the Touch Visualization Framework and the Surface Drag-and-Drop Framework. These frameworks can be used to both enhance the user experience and decrease development time. With the Touch Visualization Framework, you can reduce user error and increase their confidence and accuracy of touch interactions. With the Surface Drag-and-Drop Framework, you can let the user move content from one container to another in a rich and fluid way.

As we complete this chapter, we also complete part 2 of this book. Part 2 focused on learning WPF 4 Touch and Surface SDK, which are the main APIs that you'll need to know for any multi-touch or NUI

development in .NET. We started fairly simple but went quite deep in places, so you should be well prepared for many technical challenges you may come across.

Looking forward, we're about to start part 3, the final part of this book. In part 3, we will transition to focus on building great experiences. One factor in building great experiences is embracing the unique features of the specific devices and input technologies you may use. We'll be taking a look at a few APIs for Microsoft Surface devices that we haven't already covered as well as getting an idea of how we can integrate motion tracking sensors such as Kinect into our NUI applications. We will also consider the best way to develop multi-modal interfaces.

Part 3

Building Surface experiences

Now we can get serious about creating NUIs. We learned concepts about NUI in part 1 and how to use WPF 4 and parts of Surface SDK to build multi-touch applications in part 2. This part will put the concepts and the APIs together and show you how to build great experiences for the Microsoft Surface device.

Many of the Surface design and development challenges I describe in these last chapters can be applied elsewhere, so I would encourage you to read through this part even if you are unsure whether you will have the opportunity to develop applications for Microsoft Surface. You might get some ideas that you can use on other devices!

Part of making great experiences is creating the best interface for the specific user, task, device, and environment. In order to explore the idea of creating great experiences in more depth, we'll focus on a single device - Microsoft Surface. In chapter 11 we will discuss what types of interactions Surface is best used for and learn about the unique features of Surface. In chapter 12 we will learn additional APIs, including the complete Surface SDK, to take advantage of those features, as well as learn how to integrate and deploy applications to a Surface device.

11

Designing for Surface

Microsoft Surface is a software and hardware platform that has several unique features. The platform is designed to allow multiple people to interact with it at the same time and encourages human, social behaviors, which makes it great for collaboration. It uses vision input across the entire display surface so it can see and react to over fifty inputs at a time. It can detect the difference between your finger, a visual tag, and miscellaneous shapes that it identifies as blobs.

Surface goes way beyond a simple multi-touch digitizer that you might find on a tablet or slate. The form factor and the additional input options open up new ways to interact with a computer as well as new design challenges. We can use Surface excels in certain scenarios where it would not make sense to use a laptop or a smart phone.

This chapter discusses high-level design for Surface applications and introduces some Surface design considerations that every Surface developer should be aware of.

Even if you are unsure whether you will need to develop applications for a Surface unit, this chapter should be an interesting read that will expand how you think about other types of interactions on regular touch-screens.

11.1 Great Surface experiences

When you create an application for Microsoft Surface, you need to do more than just write code and more than just pick a few pleasant colors and shapes for the interfaces. You need to consider the entire experience.

The complete Surface experience is made up of several components, just one of which is your application. You also have to account for the background and desires of the people who will use the application, the physical configuration of the Surface, as well as the light, sound, and social environment where the Surface is deployed.

In this section we are going to discuss the Surface hardware and the components of a great Surface experience. If you have designed or developed applications for Surface 1.0, then you may be familiar with some of these ideas. Nevertheless, this section will refresh your mind with the core reasons why we love developing for Surface.

11.1.1 Overview of Surface 2.0 hardware

As I mentioned, Microsoft Surface is a hardware and software platform. We will be talking about the software in the rest of this chapter, but in this section we will get introduced to the hardware, how it can be configured, and how it works.

Microsoft Surface 2.0 has one hardware form factor manufactured by Samsung. The unit is called Samsung SUR40 for Microsoft Surface, which is a mouthful, but when we refer to the hardware itself we can just say SUR40. In this context, when we say Microsoft Surface or just Surface, it refers to the software and the hardware platform.

NOTE

In the future, it is possible that other hardware is manufactured to support the Microsoft Surface 2.0 platform, but at the time of writing no other form factors have been announced.

SUR40 integrates both a computer and display into a single form factor, shown in figure 11.1. The entire package is only four inches thin but still incorporates the vision input capability from the first version of Surface.

Figure 11.1 The Samsung SUR40 for Microsoft Surface incorporates a computer and display in a four inch thick form factor. SUR40 can be mounted horizontally on legs or vertically on a wall.

SUR40 has improved specifications over the Surface 1.0 hardware, so let's discuss what is new with SUR40 compared to Surface 1.0.

WHAT'S NEW WITH SUR40

Perhaps the most obvious new feature of SUR40 is the new form factor, which is much thinner and lighter than the Surface 1.0 hardware. This allows SUR40 to be mounted horizontally on removable legs or can be mounted vertically on a wall using a standard VESA mount.

In additional to the standard legs shown in figure 11.1, you can buy or fabricate your own legs that fit the same mounting holes. With a custom enclosure or stand, you could also mount the SUR40 at any angle. There is an accelerometer sensor inside the device that reports the current tilt through the Surface SDK to your application so you can display an appropriate interface for the current physical configuration.

In addition to the updated form factor, SUR40 also has updated technical specifications. Table 11.1 compares the Surface 1.0 hardware with SUR40.

Table 11.1 Technical specifications of SUR40 compared to Surface 1.0

Specification	Surface 1.0	Samsung SUR40
Operating System	Windows Vista Business 32-bit	Windows 7 Professional 64-bit
CPU	Intel Core 2 DUO 2.13 GHZ	AMD Athlon II X2 Dual-Core 2.9 GHZ

RAM	2 GB	4 GB DDR3, upgradable to 8 GB
Hard drive	250 GB SATA	320 GB 7200 RPM SATA II, upgradable
Connectivity	Wi-Fi b/g, Bluetooth, Ethernet, VGA out, Audio out, USB	Wi-Fi b/g/n, Bluetooth, Ethernet, HDMI in*/out, Audio in/out, USB
Graphics Card	ATI X1650 with 256 MB VRAM	AMD Radeon HD 6570M 1 GB GDDR5
DirectX version	DirectX 9	DirectX 11
Display resolution	1024x768	1920x1080
Display aspect ratio	4:3	16:9
Display type and size (diagonal)	30 inch DLP projector	40 inch LCD with Gorilla Glass
Depth	21 inches	4 inches
Display orientation	Horizontal only	Horizontal, Vertical, other tilt angles
Vision input technology	5 IR cameras with rear diffused illumination	PixelSense

* The HDMI in port allows the display to operate like a TV and display an external video source. The internal computer itself does not have HDMI input. You can switch between the computer input and HDMI input through an integrated on-screen display.

As you can see, all of the specifications have been upgraded. The updated CPU and graphics card will support modern high-end graphical and compute intensive applications. The display was also upgraded to full HD resolution at a larger 40" size and supports vision input using a new technology called PixelSense.

PixelSense is a technology invented by Microsoft that allows Surface to fit a vision input system into a four inch thin package. This reduction in depth makes it practical to mount the SUR40 vertically or at other angles. A PixelSense screen has infrared sensitive pixels embedded into the LCD display itself alongside regular RGB pixels. This enables the display to see anything placed on it, supporting massive multi-touch as well as object recognition, two key features of the Surface experience.

When you add up the integrated form factor, the upgraded specifications, and the vision input support, you can create some great experiences with Surface 2.0. In the next two sections, I'm going to go over the two key components of the Surface experience.

11.1.2 Bringing people together

Traditional computing, whether at a desktop or using a smartphone, laptop, or slate, only allows a single user to control the input devices. Other people can watch, but they are passive. Surface changes that.

People are social. Surface is social. The Surface should be thought of as something that will bring people together around the applications you create and the content you present. This is a key component of the Surface experience because humans are social beings and the Surface experience is designed to recognize and take advantage of that.

Depending upon the application, you might want to use Surface in a social meeting space such as a lobby, a social collaboration space such as a conference room, or a social learning space such as a classroom. All of these places are social and you can use Surface to enhance the social experience with appropriate information, games, and activities.

In this section, we will talk about a few of the considerations we need to keep in mind when designing social experience on Surface.

MULTI-TOUCH MEANS MULTI-USER

The Surface vision input system gives you massive multi-touch, which in practice means fifty or more touches at the same time. Naturally, a single person cannot realistically create fifty touch points, so this capability allows multi-touch to also mean multi-user.

When you design your application to be a Surface experience, you should keep in mind that multiple people may and probably will interact with the application at the same time. You should try to avoid controls and metaphors that are geared towards only a single user, such as full-screen step-by-step interactions or controls that are consolidated to within a single user's reach.

There is an exception to this guidance: if you are designing an application or part of an application where you are sure only a single user will control it at a time, then it would be appropriate to lay out the interface for that user. An example of this might be a secured administrator mode or an application with a presenter mode. Even in a presentation application, though, there might be multiple presenters.

ENCOURAGING SOCIAL BEHAVIORS

If you come from a desktop or mobile background (who doesn't, today?), then you may initially be tempted to assume that since you application runs full screen that you have the full and undivided attention of the users. This is not always the case. Since Surface applications live in a social environment, users will often go between interacting with the application and interacting with each other.

We can take advantage of these off-screen interactions between people to solve certain interaction challenges. If we account for the social customs and psychological phenomenon of our target users, we can avoid making the interface over-bearing and hand-holding the user through interactions.

As an example, consider an interaction that requires users to take turns, such as in a game. One design might involve assigning an order to users and then attempting to enforce that the correct user takes the turn. That design would be more work than it is worth and would not be received well by most users. Instead, consider adding a simple way to indicate whose turn it is and letting social obligation and the mutual desire to play fair take care of enforcing the correct person takes the turn.

Another example is helping the user learn how to use an interface. It is a good idea to make the interface as easy to learn as possible while still being functional, but certain interactions or metaphors may require a little jump-start. If you need to use visual learning clues, keep them subtle so they stay out of the way for advanced users. You should also keep in mind that when there are multiple users, they can help each other learn the interface both passively or actively. When users watch or talk to each other to learn an interface, it is significantly better experience than using an in-application walk-through or help guide.

Children helping children

One of the major Surface projects I had the pleasure to contribute to was a suite of applications for the Smithsonian Institution in Washington, D.C. The applications targeted elementary-aged children and were design to be short experiences that related to other exhibits at the various Smithsonian museums. We took advantage of the social nature of the Surface experience as well as the object recognition capability and incorporated various physical objects into the applications that were linked to the exhibit themes.

During testing at the Smithsonian Castle, we observed actual visitors using the applications on the Surface 1.0 unit. The users were mainly children, but also included teenagers, parents and other adults, and a few toddlers.

As waves of people came and left, I observed an interesting pattern in how a single child would engage over time. There were three distinct phases:

First, the child would observe other people using the applications, sometimes at a distance if it was crowded, and sometimes side-by-side if there was a spot available.

Second, the child would actively use the application. They would share control with other users, learn how to use the objects, and try each of the applications at least once, sometimes up to two or three times.

Finally, the child would take a teaching role and step back a little bit to let others use the applications, but remaining engaged enough to help other users if they didn't understand how to use a physical object or get through a particular interaction.

These observations validated our subtle, free-form design approach to the applications and illustrated the importance of the social experience while using the Surface applications. We didn't need to insert wordy instructions or hand-hold the users through interactions, and the users were more than happy to observe, interact, and then teach others what they learned.

You can view a video of InfoStrat's applications for the Smithsonian Institution here: http://www.youtube.com/watch?v=tfJu6CiygEI

It is a common mistake to try to overdesign an interface. When you can, let people solve difficult problems themselves by taking advantage of their social nature. One caveat here -- you may need to consider ways that different cultures handle social interaction. In one situation, users may be eager to help each other, but in another country, users may find it socially awkward to step in to another user's experience without invitation.

360 DEGREE EXPERIENCES

When SUR40 is deployed as a horizontal Surface, people can approach it from any direction. Even though the screen coordinate system defines a top, bottom, left, and right, you need to think beyond those labels to create 360 degree experiences. If you are creating applications at your desktop using a vertical monitor, it is even more important to keep this in mind.

When you are creating a 360 degree interface, you need to make sure the interface is usable from any position. This means that people should be able to read the text content and reach controls regardless of where they are. Content may not always be properly oriented to every user, but it should be easy to rotate the content as the user desires.

One way to achieve this is to organize everything in the application in a ScatterView. This might work for certain applications, but for others the ScatterView can become too disorganized and is not an appropriate container metaphor. For those other applications, you should design the interface such that it can easily be reoriented. The Surface launcher, shown in figure 11.2, has a top and bottom but allows users to flip it around to face the other side of the table.

Figure 11.2 Top: The Surface Launcher provides an simple way for users to find and launch applications using a variation on a horizontal scrolling area. Bottom: The Launcher can be reoriented to face either side of the table. The handles on the sides allow the user to pivot the entire launcher around the center until it flips completely around to face the opposite site. Due to the aspect ratio, there is not enough room to drag the handle completely around, but the launcher will flip if you flick or drag the handle to the edge of the screen.

Another factor in 360 degree interface design is choosing appropriate starting positions for content. If you know the approximate location of the person who requests content then you can intelligently orient and position the content so it is ready to be used without unnecessary manipulations.

One way to determine a person's location is while your application is starting, check the value of the ApplicationServices.InitialOrientation property. This property tells you which way the Surface launcher was oriented when your application was launched. We will cover this property and others in more detail later in this chapter.

Another way to determine a person's location is by checking properties on touches. The position and orientation of the touch can suggest good positioning for new content.

A final way to orient new content to a user is to take advantage of other interface elements that have already been oriented. Let's say a user has moved a ScatterViewItem to face them so they can search for a document. It would make sense to display the search results near the search query interface using approximately the same orientation.

HORIZONTAL AND VERTICAL EXPERIENCES

A final consideration when designing Surface applications to bring people together is to consider how your application may work on both horizontal and vertical displays, as well as other tilt angles. While some applications may make sense deployed only to horizontal displays or only to vertical displays, other applications could target multiple tilt angles with minimal effort. Keep in mind as well that if you are planning to run the application on Surface as well as regular Windows 7 touchscreens, the touchscreens are considered vertical displays by default.

We already discussed some considerations about designing 360 degree interfaces. When you support a vertical display, you'll want to modify the interface in subtle ways. Instead of people approaching the device from any position, you now have a few people that may stand side-by-side while facing the display.

The same main point applies here: orient the content towards the users and put interface elements within a comfortable reach. You also need to make sure that content is not appearing upside down to users, so you may need to modify any randomization algorithm used for placing and orienting content, as well as any background text or images that might end up upside down to people using a vertical display.

There may also be a crowd of passive observers standing back from the SUR40 while others use it. You can address their needs by displaying high-level shared information near the top of the display where other people's bodies are unlikely to obscure content. You may also want to display this information in a larger size that is easier to read from farther away.

This is only a quick overview of some of the user experience consideration, but it should get you started in thinking about how to design interfaces that bring people together.

In the next section, we will discuss the other main component of the Surface experience: object recognition. After that, we will walk through a person's typical Surface experience from first touch to using the Surface launcher.

11.1.3 Interacting with physical objects

Physical object interaction is a unique feature of Surface and is not found on many other commercially available displays. The vision input provided by PixelSense allows Surface to see not only fingers and hands but other objects placed on the display. These objects can be used as a part of the interface. You can design Surface applications to use both touch and objects to create intuitive and engaging experiences.

The most typical and reliable way for Surface to recognize an object is attach a tag to it. Tags are visual patterns that are oriented and encode a numeric value. You can use the tag position and orientation to display interface elements around the physical object, and you can use the tag's value to uniquely identify specific objects. Figure 11.3 shows a typical tag that Surface can recognize.

Figure 11.3 Microsoft Surface can recognize tags and identify their value, position, and orientation. Shown here is a byte tag, which encodes an 8-bit value. Surface 2 also supports another tag format with a higher number of possible values, not pictured here.

Surface also gives you access to the raw image that the Surface sees. For certain applications, it might make sense to do processing on the raw image to recognize shapes or other patterns. For example, if you want to use several types of physical objects with different shapes, but it was impractical to attach physical tags, you could use the raw image to find and track the unique shape of each object.

WHY OBJECTS?

You may ask, "What is so great about recognizing physical object?" or "Why bother with a physical object when I could create a digital control?" These are valid questions and get to the heart of why the Surface experience is different from a regular capacitive multitouch screen. There is a simple reason why you should consider using physical objects in your application:

Objects bridge the physical and virtual worlds.

Even the most responsive touchscreen does not compare to the intimacy of touching a physical object. The object is real. There is no lag. It has a shape and a texture and we enjoy how it feels to interact with it. We can manipulate the physical object while our eyes are looking elsewhere. We feel more precise and confident using a physical object.

We can package up all of these visceral reactions into a seamless magical experience. Using objects creates an experience that is more engaging but often more efficient than a pure touch interface.

I want to spend a little more time discussing objects so in the next section we will learn some patterns that you can use when designing object interactions

11.2 Object interaction patterns

There are many things you can do when an object is placed on the Surface. In this section I'm going to break down a set of object interaction patterns that I have identified. This list is based upon existing object interactions in real Surface applications, but is not exhaustive. These patterns are also not exclusive. You can, and often will, combine multiple patterns within a single object interaction. My hope is that these patterns help to inspire you about different ways to incorporate object interaction in your applications.

In each of these patterns, I will give a definition, some examples and a screenshot, if available, and comments about the pattern.

11.2.1 Perceived affordance

The perceived affordance pattern states that the physical form of the object helps us understand how to use it with the application. In this case, replacing one object with an object with a different shape would hurt the usability of the application because a different shape may give the user no clue or a false clue about how the object should be used. Figure 11.4 shows several examples of objects that use affordance.

Figure 11.4 A suite of InfoStrat applications for the Smithsonian Institution. Top: a flashlight virtually illuminates an undersea world. Left: A paintbrush leaves stroke marks in a painting application. Right: A physical stick is spun on a virtual log pile to start a fire. The form of each object gives hints to the user about how to use it with the application. Note that the flashlight interaction worked on Surface 1.0 but due to changes in how the input system works may not work with a 2.0 device.

This pattern does not apply to objects where the form is simply used for identification and don't give clues about the interaction, such as tagged products used to pull up information about the product.

The affordance may be perceived in combination with the virtual interface. A triangle or square object would not have any perceived affordance by itself, but if the interface also had a triangle or square shaped area indicating appropriate locations for the objects, the object shape does have affordance.

11.2.2 Augmented objects

An augmented object is one where when the object is placed on Surface, it is augmented by animations and visuals that bring it to life or give it meaning in a larger context. Figure 11.5 shows an example of an augmented object.

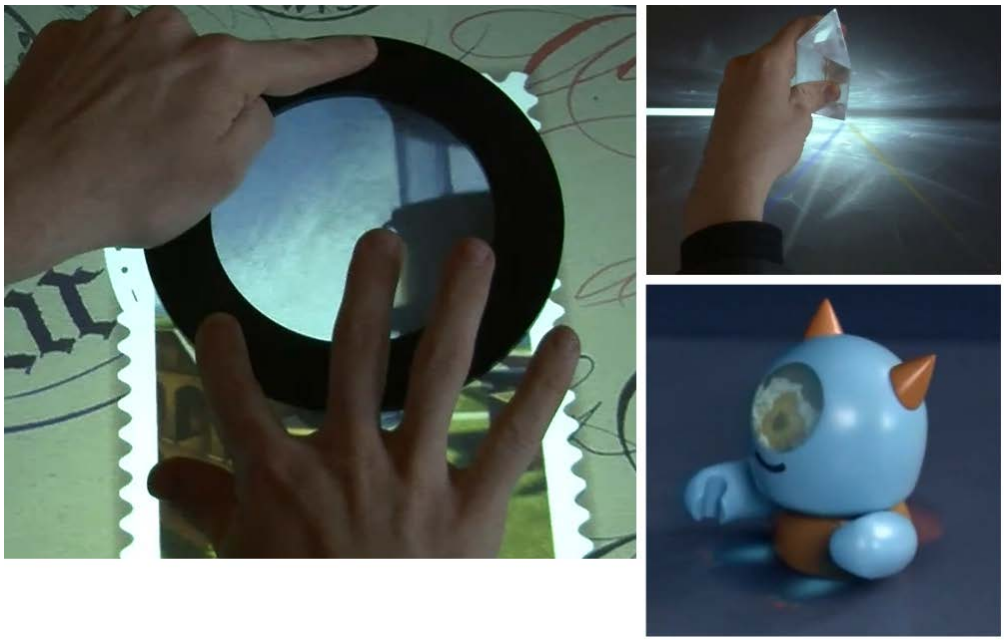


Figure 11.5 Left: The physical ring virtually magnifies whatever is inside of it. Here the user zooms in on the image of a lighthouse. (InfoStrat) Top: A physical prism placed in the path of a virtual light beam cause the light to split and refract into different colors of the rainbow. (InfoStrat) Bottom: The Surface Monster comes to live when placed on the Surface. (Microsoft)

Other ideas for augmenting physical objects include creating laser table or wind tunnel simulations around physical models, a virtual character that explores a flat physical object like a house, or a physical visor that reveals hidden information to one user while blocks it from others.

11.2.3 Content association

With the content association pattern, the object allows the user to access specific content associated with the object. This might work well in a retail scenario where tagged products are placed on Surface to pull up content specific to the product. It could also be used for presentation software to allow users to prepare specific objects that display specific content.

The Surface shell enables a variation of content association called object routing which allows tag values to be associated with specific applications. Placing a configured object routing tag value on Surface will let the user to launch the application if he or she clicks on the displayed application icon, even if the user is located

11.2.4 Controller

The controller pattern means that the position, orientation, or mere presence of the object controls changes variables in the virtual world. Examples of this might include placing and rotating a cylinder to

pick a color or other items in a list, or an object that controlled certain axis of a 3D manipulation. Another example would be placing an object to apply a filter or highlight to information when it is placed on Surface.

11.2.5 Device interaction

With device interaction, the tag identifies another computing device. Combined with another communication channel such as Wi-Fi or Bluetooth, information can be exchanged and transferred seamlessly out of the physical device and into the Surface's virtual world, or back from Surface to the device. It should be noted that the communication channel and the association between specific device and the tagged value needs to be configured ahead of time.

11.2.6 Emergent group

The emergent group pattern is used when multiple objects from a logical group of objects are placed on Surface, the behavior changes and there are emergent effects. Initially, this pattern may seem like multiple instances of the controller pattern, but what is unique about this pattern is that interface reaction changes based upon the number of objects placed and their proximity.

Consider the item compare Surface SDK sample application. When a single object is placed, product information is displayed. When a second object is also placed, the two products show a side-by-side comparison view. This pattern relies on the fact that the physical objects are part of a single logical group, such as products in this case.

11.2.7 Off-screen information

Off-screen information means that the object has text or visuals that provide extra information outside of the Surface's display about the object or how to use it. This could be as simple as a label or icon representing a category of information associated with the object or as involved as a tag printed on a brochure with instructions on visiting a store. This pattern can be combined with device interaction if the device dynamically displays extra information about the interaction or the content.

11.2.8 Optics

The object has special optical properties that result in non-traditional input and output, including transforming the display output or providing specialized input. We revisit the Surface Monster in figure 11.6.

Figure 11.6 The Surface Monster uses a bundle of fiber optics to bend the animated eye displayed underneath the toy into position on the face. This works both ways and the application can detect and react when you poke the eye. (Microsoft)

Other applications of optics include Lumino blocks by Baudisch et. al 2010 which are stackable block and cylinders where Surface can identify individual blocks, and Surfaceware by Microsoft which allows Surface to determine when a glass is almost empty.

11.2.9 Tagless identification

With tagless identification, the raw image is used to identify the object by shape or by IR image pattern. Examples of this include book cover recognition, leaf identification, and Lumino blocks.

11.2.10 Tracking

Perhaps the most commonly used and combined pattern, tracking allows visual elements to be positioned relative to the object and follow its position and orientation.

Now that we have discussed the two key components of the Surface experience and learned several object interaction patterns, bringing people together and object interaction, let's talk a bit more about what people actually sees and do when they use a Surface.

11.3 Walking through a session

In order to help you understand the flow of the Surface experience, I'd like to walk through a typical session at a Surface unit. In this section, I will focus on the Surface Shell, which includes everything the user sees or interacts with that is not an application. The Shell includes, attract mode, the application launcher, the access points, and provides a few other services such as the Surface keyboard, notifications, session time outs. In addition to talking about what the Shell is, I'll also describe some of the reasons behind the design of the Shell, which you can hopefully apply to your applications to create a consistent Surface experience.

11.3.1 Attracting users

When Surface first boots, or no one is using Surface for a period of time, it will enter attract mode. Attract mode is a special screen that encourages people to touch the Surface and learn how to interact with it. Figure 11.4 shows a screenshot of attract mode.

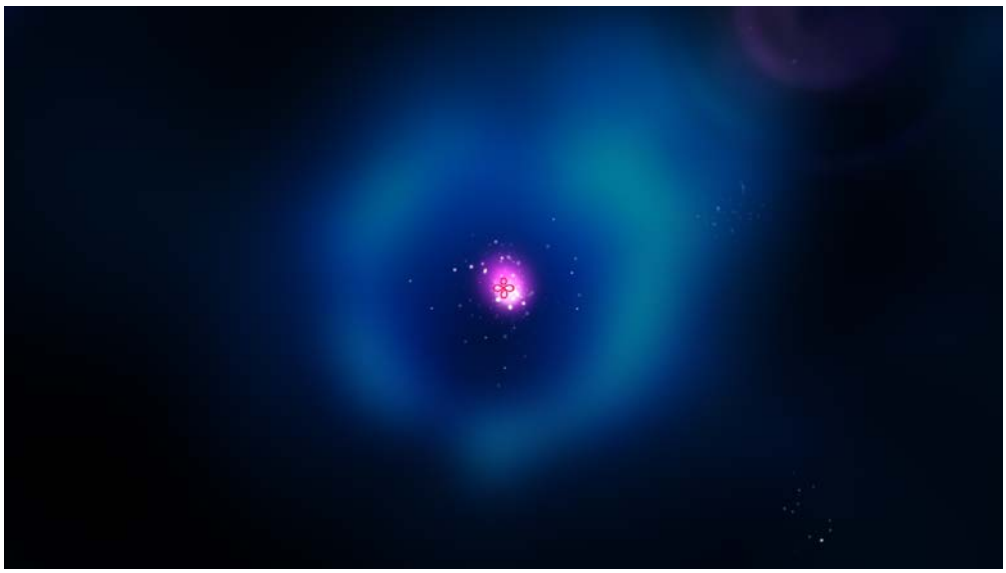


Figure 11.7 Attract mode uses simple animations and interactions to encourage people to touch Surface and learn how to use it. You can see here a water ripple in response to touch and generated glowing particles that accelerate towards the center button.

When someone touches a Surface in attract mode, it creates ripples in a water simulation and also generates glowing clouds of particles that slowly accelerate towards the center of the display where the Surface logo is. When no one is touching a Surface in attract mode, it will randomly trigger ripples and particles. This shows people what Surface can do and entice them to touch it themselves.

The highly-responsive water physics and the mysterious particles start to give people the idea that Surface interactions involve play and exploration. Once someone takes the first step and understands that Surface responds to touches, they can start to discover the rest of the Surface experience.

NOTE

The design of attract mode is a perfect example of a learning task that we discussed way back in chapter 1. You must first learn and demonstrate that you know you can touch Surface in attract mode before it will grant you access to the more complicated launcher task. Attract mode successfully introduces novices to the idea of touching the Surface while not being a burden on experts, who can simply tap the Surface logo to open the launcher.

The particles accelerating towards the Surface logo draws attention to the logo and encourages users to touch it. When someone touches it, attract mode will transition into the launcher and a Surface session will be started.

A Surface session starts when the launcher is opened and ends when Surface returns to attract mode, either by user action or time out. During a session people may start and interact with several applications and switch between applications using the launcher.

11.3.2 Launching applications

The Surface launcher provides a way for people to explore and launch available applications. It also lets people switch between running applications. Applications are presented as square images with an application name above and a description below. The applications are within a horizontal scrolling area called the launcher. We previously saw the launcher in figure 11.2, and figure 11.8 below shows how the launcher reacts after the user taps on an application to launch it.

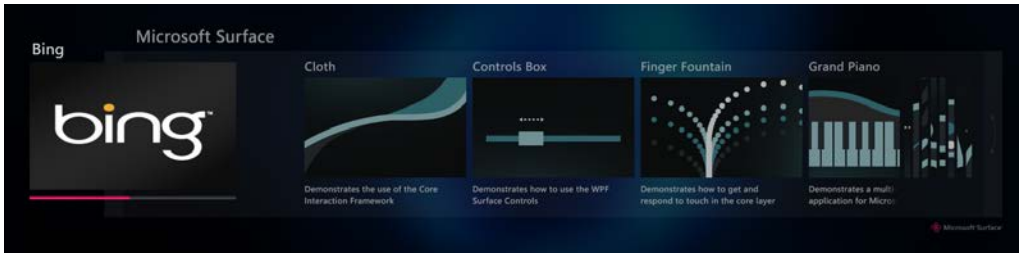


Figure 11.8 The Surface launcher allows people to explore and launch the available applications. Applications are represented using application icons with the application name displayed above the icon and a short description below the icon. Here the user has tapped the Bing application icon and Launcher shows the progress of launching the application with a progress bar. The currently loading application grows and the rest of the launcher shrinks and fades slightly.

The Surface 2.0 launcher differs from the Surface 1.0 launcher in two important ways. First, instead of scrolling to an infinite looping list of applications, the Surface 2.0 launcher displays all applications in a single area. When there are more than five applications, the application icons shrink down at the left and right edges of the launcher into a thin vertical area. If you tap one of the thin application icons at the edge of the launcher it will jump directly to that application. Figure 11.9 shows an example of the shrinking application icons.

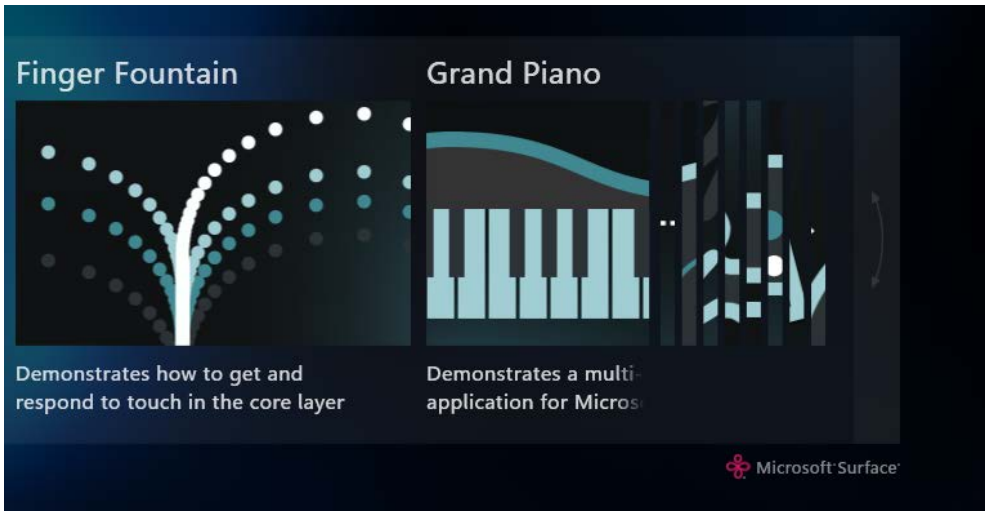


Figure 11.9 As the launcher is scrolled, application icons smoothly shrink down to a thin vertical space at the edge, shown in the right of this image. This gives the user some context about where they are in the virtual space of all applications and also allows the user to tap on a specific application to jump to it.

The second way that the Surface 2.0 launcher is different from 1.0 is that you can rotate the entire launcher around its center so that it will face the other side of the table. This 360 degree design replaces the Surface 1.0 method of orienting the launcher to whoever last touched an access point.

You can rotate the launcher by dragging one of the labeled edges of the launcher towards you and releasing it near the edge of the screen, where the launcher would continue to rotate around 180 degrees to face the other side of the table. We saw an example of this earlier in figure 11.2.

Once someone has picked an application, they simply need to tap on it to launch it. A small progress bar will show the status of the application as it loads and in a few seconds the application will be displayed.

Object routing

In addition to the launcher, there is also a way to launch an application using a physical object. During attract mode or at the launcher, if the user places a tagged object on Surface and the value of the tag is associated with one or more applications, then the user will be presented with an option to launch those applications. This is a good shortcut for accessing commonly used applications.

Applications run full screen, so the launcher will no longer be visible. In order to get back to the launcher, people need to discover how to use the Surface access points.

11.3.3 Activating access points

Surface features four access points, one in each corner, that allow users to navigate from an application back to the launcher, or from the launcher back to attract mode. Navigating from the launcher to attract mode using the access point ends the current Surface session and closes all running applications. The access points are always present except in attract mode and always displayed on top of any other graphics.

The design of the access points in Surface 2.0 is different than Surface 1.0. Surface 1.0 access points were quarter circles in the corner that acted like buttons. They suffered from accidental activation and for certain users were difficult to figure out without watching someone else use them. In Surface 2.0, the access points are full circles and activate using a sliding interaction, which practically eliminate accidental activation. When you touch one of the new access points, it displays text and visual hints about their purpose, as shown in figure 11.10.

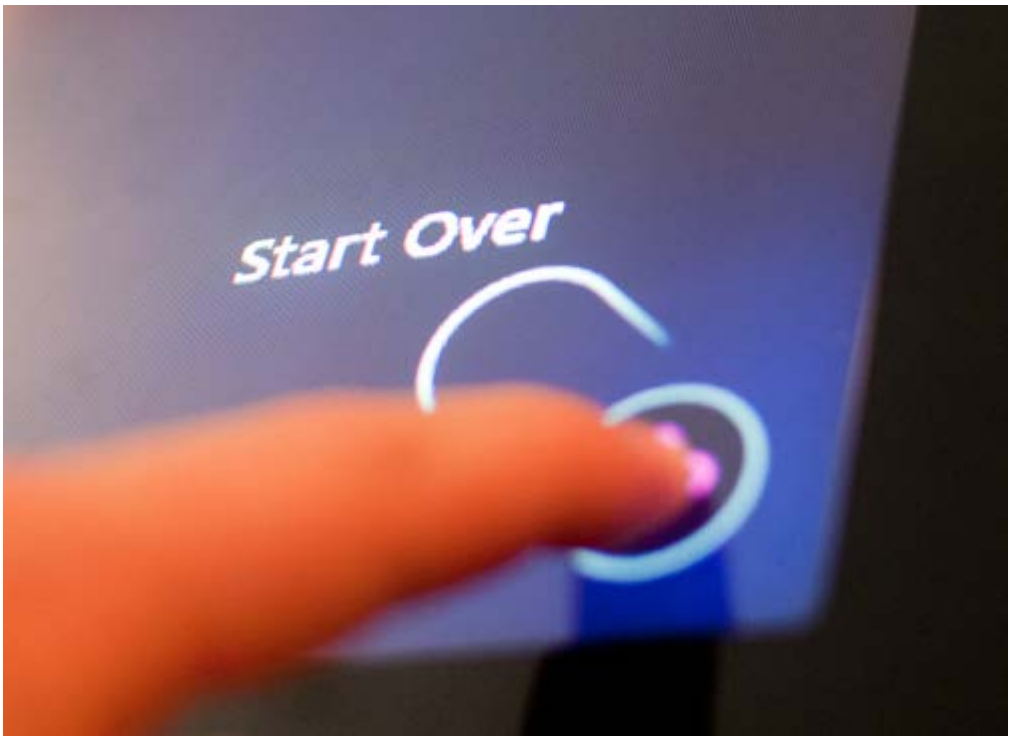


Figure 11.10 The Surface access points are located in each corner. Normally it just displays the logo, but when touched, the access point displays a visual cue that you can drag the point inward. This access point is displayed during the launcher and the text hint invites the user to start over. If this access point were slid into the curved area, Surface would switch to attract mode.

Another difference between the Surface 1.0 access points and Surface 2.0 access points is that in Surface 1.0, the access points were used to toggle between attract mode and the launcher, as well as go from an application to the launcher. These multiple meanings were not apparent to new users and caused some confusion.

The Surface 1.0 access points were also not completely apparent to casual users, which resulted in many people playing with the Surface 1.0 attract mode and then walking away but never realizing there was more to the experience. Surface SDK 1.0 SP1 attempted to improve this by adding an animation to display the access points when someone first touches attract mode, but the animation was in peripheral vision and often missed.

The challenges seen by the Surface 1.0 attract mode and access points are solved with the new Surface 2.0 attract mode and access point design. You can see now why Surface 2.0 attract mode has the particles that animate towards a central Surface logo button. This new design puts the next step of the experience directly in the center of your vision and draws your eyes toward it with the motion of the particles.

Attract mode, the launcher, and the access points are the core parts of the Surface 2.0 experience, outside of the applications themselves, of course. While interacting with Surface, you may come across a few other features of the Surface Shell such as the Surface keyboard, notifications, and the time out screen. I just want to briefly go through each of these features.

11.3.4 Typing with Surface keyboard

The Surface keyboard is displayed whenever you touch a SurfaceTextBox or SurfacePasswordBox. The keyboard has visual responses to key presses for SurfaceTextBox, but the visual feedback is suppressed for security reasons when the target is a SurfacePasswordBox. The keyboard has two layouts, full and numeric, which you can select control programmatically based upon which is more appropriate for the

context. We will learn the Surface keyboard APIs later in this chapter. The full Surface keyboard layout is shown in figure 11.11.

Figure 11.11 The Surface keyboard as used in the Bing for Microsoft Surface application. The keyboard can be used for text and password entry and can be rotated towards any user.

Only one Surface keyboard can be displayed at a time, which means that text input can only be performed by a single user at a time. One reason for this limitation is that if there were multiple keyboards, there could be confusion about which keyboard is sending input to which control. There is also an alternate version of the Surface keyboard for numeric input, shown in figure 11.12.

Figure 11.12 The Surface keyboard can also be changed to a numeric layout, appropriate for entering pin numbers or other numeric data.

The Surface keyboard is a feature of the Surface Shell and is only available on a Surface unit. When testing applications using the Surface Input Simulator or applications that target Windows 7, you will need to use a real keyboard or the Windows 7 virtual keyboard.

11.3.5 Notifications

Notifications allow background applications, service applications, or the Surface Shell to display a message to users. A notification consists of a title, message, image, and image caption displayed in a pop-up bar on the bottom of the screen. Users can dismiss a notification by clicking the close button, or the notification will disappear after a configurable amount of time.

There are type types of notifications: informational and active. Informational notifications simply provide users with information and can only be dismissed or time out. Active notifications are similar but also allow users can tap on the message or image to launch the associated application. Figure 11.13 shows an active notification.

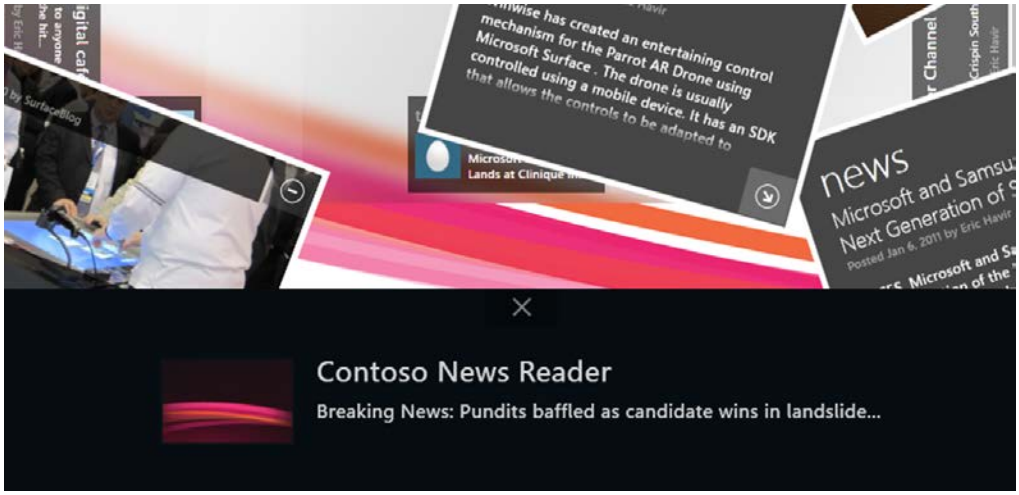


Figure 11.13 A notification from a news reader application displayed while a different application is active. This notification

is displayed across the entire side of the Surface and prompts the user to switch to the news reader application.

Another use of notifications is when the user tries to launch an application but it crashes or does not complete initialization within a timeout. Rather than display a crash dialog typical in Windows or make the user wait an extraordinary amount of time, the Surface Shell will close the application and display a user-friendly message that the application is not working.

11.3.6 Session timeouts

There are several configurable timeouts that the Surface Shell enforces. If there is no input activity for a period of time, the Shell will display a screen that prompts the user to continue the current session or start a new session. If a choice is not made for a period of time, Surface will end the current session and return to attract mode.

These timeouts are designed primarily for when Surface is deployed for public use. When there is a constant flow of people using Surface, they can continue the same session, but if there is a long enough gap between groups of people, Surface will reset to attract mode to present a fresh experience to new users. The session timeout screen accounts for the possibility that people are still using Surface but are occupied and have just not touched it for a while.

Now that we have a good idea of what Surface is and its unique capabilities, we can move on to learn about Surface-specific APIs and controls as well as how to integrate applications with the Surface shell.

11.4 Summary

In this chapter we were introduced to the next generation of surface computing devices, the Samsung SUR40 for Microsoft Surface. We learned about how SUR40 compares to Surface 1.0 and the unique aspects of the Microsoft Surface experience, including bringing people together and interacting with physical objects. We also saw several examples of object interaction patterns and learned about the different features of the Surface Shell.

Surface UX Guidelines

In this chapter I gave a high-level overview of the Surface experience and some specific ideas for designing Surface applications, particularly with object interaction. The Surface team has also published specific User Experience Guidelines that I would encourage everyone to read. These and other training materials are available from http://surface.com in the technical resources section.

While there are many different types of touch devices available, from desktop LCDs with 2-touch IR frames to slates and tablet with capacitive digitizers to larger LCD touch screens, Microsoft Surface stands out as the best touchscreen on the market. It supports massive multitouch and multiple users in both horizontal and vertical scenarios while also supports tagged objects and gives access to the raw vision input. It combines these capabilities with an integrated Surface Shell experience and a world-class SDK to provide the best user experience and best developer experience.

The SUR40 hardware is more capable and versatile than Surface 1.0 while costing almost half of the old price. While the price may still be outside of certain budgets, it is a great value when compared to other similar hardware and is appropriate for use in a wide variety of industries from public sector and education to retail, marketing, and medical applications to integration with line-of-business systems and enterprise collaboration.

Microsoft Surface is a platform for innovating the next generation of natural user interfaces.

In the next and final chapter of this book, we will learn about how to develop, integrate, and deploy applications to Surface as well as controls and APIs unique to Surface.