

# Implementing multi-user multi-touch scenarios using WPF in Windows\* 8 Desktop Apps

## Summary

In this paper we walk through a sample application (in this case a game that quizzes people on the Periodic Table) that enables multi-user, multi-touch capability and is optimized for large touchscreen displays. By using User Controls and touch events, we can enable a scenario where multiple users can play the game at the same time.

Windows Presentation Foundation (WPF) provides a deep touch framework that allows us to handle low-level touch events and support a multitude of scenarios from simple touch scrolling to a multi-user scenario. This game has two areas where users can touch, scroll, and click using their fingers simultaneously while the remainder of the UI remains responsive. Finally, this application was designed and built using XAML and C# and follows the principles of the Model-View-ViewModel software development pattern.

## Supporting Large Touch Displays and multiple users in Windows Presentation Foundation

WPF is an excellent framework for building line-of-business applications for Windows desktop systems, but it can also be used to develop modern and dynamic applications. You can apply many of the same principles you use for designing applications in WPF with some small tweaks to make them friendly and easy to use on a large format display.

The XAML markup language has, as a foundational principle, **lookless** controls. This means that the appearance and styling of a control is separate from the control's implementation. The control author may provide a default style for the control, but this can easily be overridden. If you place a style in your XAML (inferred or explicit), it will append the base style that ships with the framework. You can also use the template extraction features in Visual Studio\* 2012 to make a copy of styles and templates that ship with the .NET framework to use as a starting point.

Let's look at an example:

To create a window with a custom close button, I created an empty WPF project in Visual Studio and edited the MainWindow.xaml file as follows:

```
<Window x:Class="ExampleApplication.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" WindowStyle="None">
    <Grid>
```

```
<Button HorizontalAlignment="Right" VerticalAlignment="Top" Content="Close"
Click="Close_Window" />
</Grid>
</Window>
```

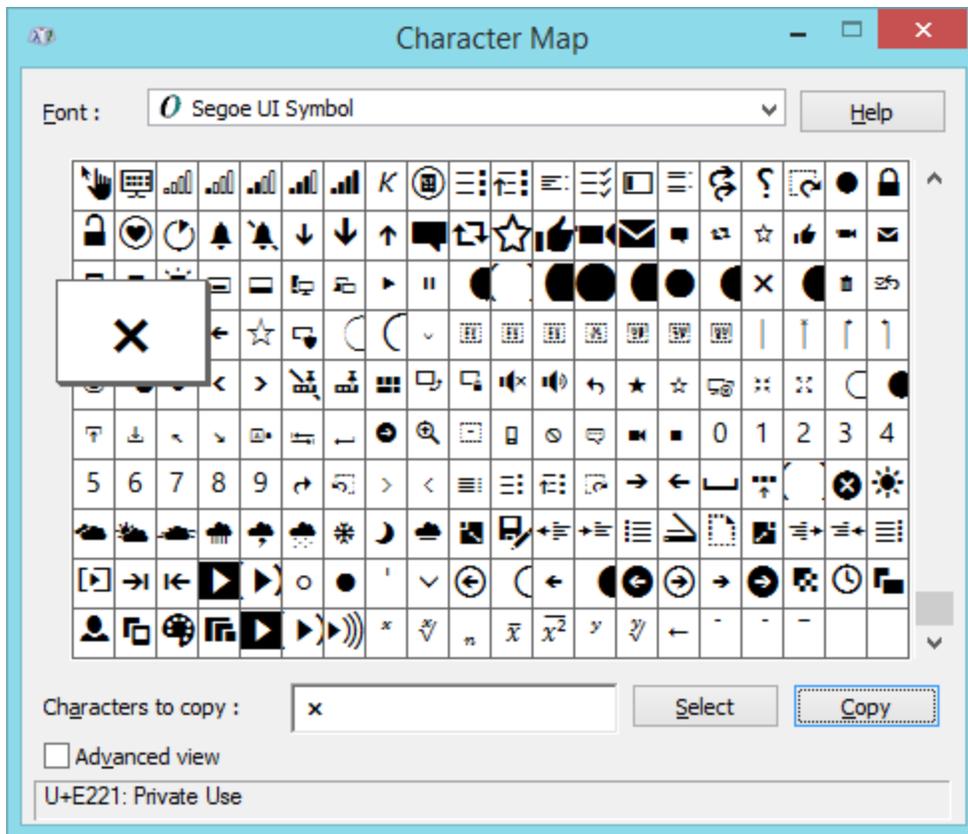
I then wrote a C# method to handle closing the window:

```
private void Close_Window(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

This created a Window like the one below:



Since we are on the Windows 8 platform, we can use the Segoe UI Symbol font to put the close symbol in the button. You can browse for the symbol you want to use in the Windows Character Map under the Segoe UI Symbol font:



Now that I have the character code, I can begin customizing the button. To start, I added the close symbol to the button:

```
<Button HorizontalAlignment="Right" VerticalAlignment="Top" FontFamily="Segoe UI Symbol" Content="&#xE221;" Click="Close_Window" />
```

I also want to style the button to make it touch-friendly by applying an XAML style. This can be done by creating an inherit style that is anywhere above the button in its Visual hierarchy. I will add the Button style to the Window's resources so that it's available to any button within the Window:

```
<Style TargetType="Button">
    <Setter Property="BorderBrush" Value="White" />
    <Setter Property="Background" Value="Transparent" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="BorderThickness" Value="2" />
    <Setter Property="Padding" Value="12,8" />
    <Setter Property="FontSize" Value="24" />
    <Setter Property="FontWeight" Value="Bold" />
</Style>
```

To illustrate this effect, I changed the Window's background color to white. The above style will result in a button that appears like this:



You can always change the style to have a larger icon and less padding, for example. With buttons and text content, you may find yourself using static padding, margin, and size values since they rarely change. If you want text content to be truly responsive, you can always put text content in a ViewBox so that it scales in size relative to the Window. This isn't necessary for most large-screen applications, but it is something to consider if your application will operate at very extreme resolutions.

For most UI elements, you will want to base your padding and margins off of relative sizes. This can be accomplished by using a Grid as your layout system. For example, in the demo application, we wanted a very thin amount of space around each periodic table element. I could use a 1px padding around each item, but the appearance of the width of that padding will differ between users on large displays and small displays. You also have to consider that your end users might be using much larger monitors and resolutions than your development environment may support. To resolve this issue, I use the grids to create rows and columns to represent the padding. For example, I can create a grid with 3 rows and 3 columns like below:

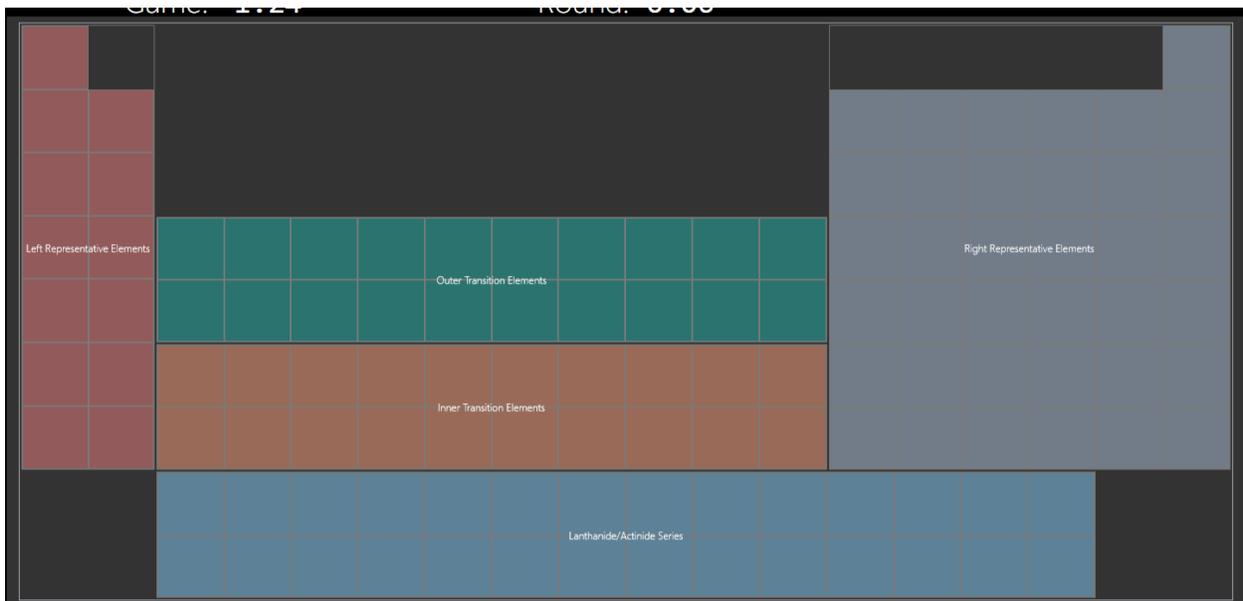
```
<Grid x:Name="tableRoot">
  <Grid.RowDefinitions>
    <RowDefinition Height="0.01*" />
    <RowDefinition Height="0.98*" />
    <RowDefinition Height="0.01*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="0.01*" />
    <ColumnDefinition Width="0.98*" />
    <ColumnDefinition Width="0.01*" />
  </Grid.ColumnDefinitions></Grid>
```

In grid definition sizing you have three options available. You can do static sizing using an absolute height or width, auto sizing that depends on the content to measure and determine size or relative

sizing or you can mix and match the different options. In our example, we make heavy use of the relative sizing. The XAML engine sums the values for the relative sizing and assigns sizing that is equivalent to the ratio of that individual value to the whole. For example, if you have columns sized like below:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="4*" />
    <ColumnDefinition Width="7*" />
    <ColumnDefinition Width="9*" />
</Grid.ColumnDefinitions>
```

The sum of the column widths (4, 7, and 9) is 20. So each width is the ratio of each value to the total of 20. The first column would be 4/20 (20%), the second column would be 7/20 (35%), and the final column would be 9/20 (45%). While this works fine, it's considered a good practice to have all of your columns (or rows) sum up to either 100 or 1 for simplicity's sake. In the first example, we make sure that the heights and widths add up to a value of 1. The column and row indexes are zero-based so we can put the content in Column 1 and Row 1 and it will have a 1% padding all around. This is 1% regardless of the resolution and will appear relatively the same to users regardless of their resolution. A padding set to a static size will be much thinner on a large touchscreen display with a high resolution than you expect it to be during development. In the periodic table application, you can see this 1% padding when browsing the table itself:



You can also enable touch scrolling for your application to make it more responsive. Out of the box, WPF allows you to use your finger to scroll through a list element. The ScrollViewer does lock your scrolling to each element so it's more like flicking between elements. If you want to enable "smooth" scrolling, you should set the **PanningMode** of the ScrollViewer. By default, the PanningMode is set to None. By setting it to **VerticalOnly** or **HorizontalOnly**, you will enable smooth scrolling through items in a list view. In the Periodic table application, the **ScrollViewer.PanningMode** attached property is used to enable this scenario on a typical ListView. I also set the **ScrollViewer.CanContentScroll** property to false so that the items will not snap and the user can use their finger to hover between items.

```
<ListView x:Name="SecondBox" Background="Transparent" ItemsSource="{Binding
Source={StaticResource PeriodicData}}"
    ScrollViewer.VerticalScrollBarVisibility="Disabled"
    ScrollViewer.HorizontalScrollBarVisibility="Visible"
    ScrollViewer.PanningMode="HorizontalOnly"
    ScrollViewer.CanContentScroll="False"></ListView>
```

The ListView mentioned is used in the application for viewing Periodic table items like below:

Outer Transition Elements							
Sc Scandium 21	Cr Chromium 24	Co Cobalt 27	Zn Zinc 30	Nb Niobium 41	Ru Ruthenium 44	Ag Silver 47	He Helium 2
Ti Titanium 22	Mn Manganese 25	Ni Nickel 28	Y Yttrium 39	Mo Molybdenum 42	Rh Rhodium 45	Cd Cadmium 48	B Boron 5
V Vanadium 23	Fe Iron 26	Cu Copper 29	Zr Zirconium 40	Tc Technetium 43	Pd Palladium 46		C Carbon 6

Finally, WPF allows us to use the built-in touch support that has been around since Windows 7. Windows recognizes touch input as a mouse when you don't specifically handle the touch events such as **Tapped**, **ManipulationDelta**, and **ManipulationEnded**. This allows you to handle the event where users tap any of the above items by using the **Click** event handler. This also minimizes the amount of code necessary to support both touch and a mouse.

Since touch support is implemented on a very low-level, the WPF platform does not group touches by user or clusters. To get around this, you typically see control authors use visual cues (such as a border or a box) to indicate to users that they should touch within a specific area. To support multiple users, we can put the touch-supported controls within a UserControl. The browsable Periodic table that is used to find the Periodic elements as part of this game is a UserControl so we can put as many or as few as we want on a screen by putting the logic into a UserControl.

## The Model-View-ViewModel Pattern

When building the application, it would be easy to write the code in the xaml.cs file and call it a day, but we want to maximize code reuse and build an application that is truly modular. We can accomplish this by leveraging the MVVM design pattern. In the Periodic Table application, every screen is bound to a ViewModel. This holds information for data-binding and controls the behaviors of the different Views. We also have a data source that uses XAML and need to manipulate the data source to run the game. The data source will be discussed in greater detail later in this article.

Since MVVM is a popular design pattern, it is possible to use it in the WPF, Windows Store, and Windows Phone platforms. To support this scenario, we can put our Models and ViewModels into

Portable Class Libraries (PCLs) that can be referenced by all of those platforms. The PCLs contain the common functionality and namespaces between all of those platforms and allow you to write cross-platform code. Many tools and libraries (such as Ninject, PRISM's EventAggregator, and others) are available via NuGet and can be referenced in a PCL so you can create large-scale applications. If you need to support a new platform, you simply create new Views and reference the existing ViewModels and Models.

This application is parsing a static data file that contains information about how to render the Periodic table. The Models are aware of the classes in WPF so PCLs would not be appropriate in this example.

In this application, we use the PRISM framework to leverage the already well-built modules for MVVM development.

For the home page, we have a BaseViewModel that has one command. The ExitCommand closes the application when executed. We can bind this command to the button mentioned earlier in the article by applying a data binding to the Button's **Command** dependency property.

```
public class BaseViewModel : NotificationObject
{
    public BaseViewModel()
    {
        this.ExitCommand = new DelegateCommand(ExitExecute);
    }

    public DelegateCommand ExitCommand { get; private set; }

    private void ExitExecute()
    {
        Application.Current.Shutdown();
    }
}
```

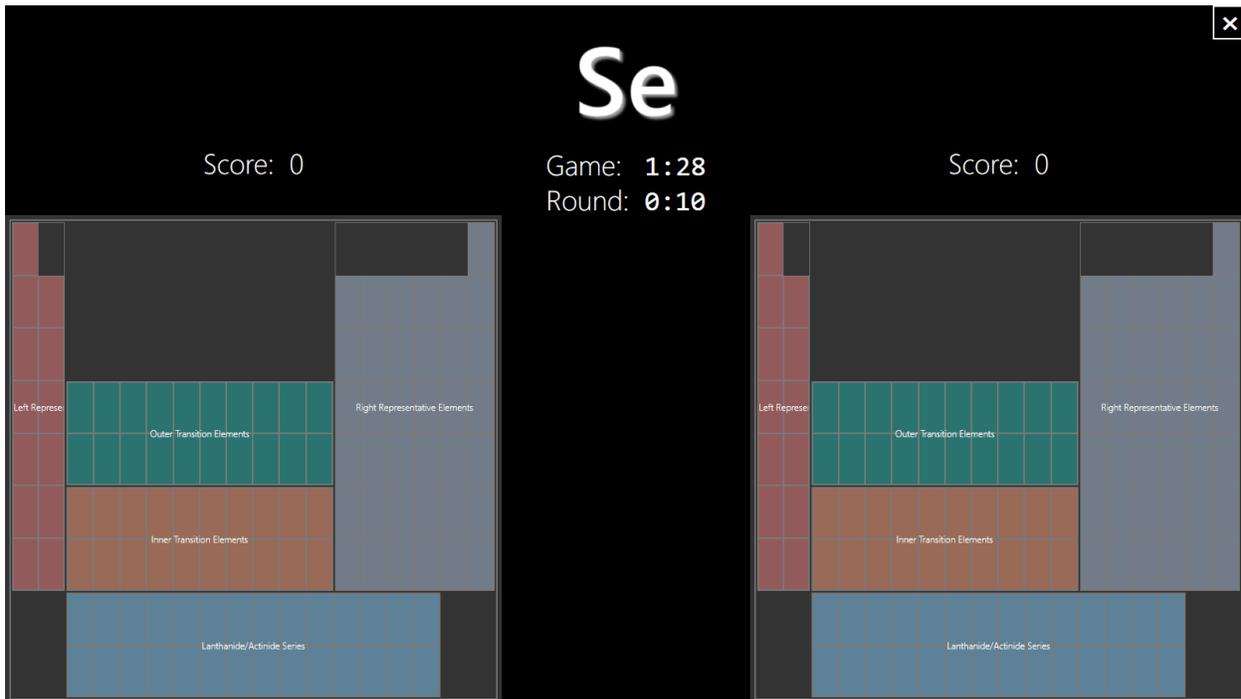
First, the ViewModel inherits from PRISM's NotificationObject class. This class contains all of the logic to let the View know when a ViewModel's property is updated. This is accomplished by implementing the INotifyPropertyChanged interface. If you ever want to look at a very solid best-practices implementation of INotifyPropertyChanged, view the source code for the PRISM project to see how the team at Microsoft implemented the interface.

Next, we use the DelegateCommand class from the PRISM framework. DelegateCommand is an implementation of the ICommand interface that is the heart of commanding in WPF. This class can be used to handle a button's click event and the logic for determining whether a button is enabled. This support not only applies to buttons, but is the primary case when the ICommand is used.

In our BaseViewModel class, we create a new instance of the DelegateCommand class and pass in the ExitExecute action to be executed when the Command is invoked (by pressing the button).

Because you can close the application from any screen, all of the other pages inherit from the BaseViewModel class. To keep all of the game-related logic together, both the 1-player and 2-player games use ViewModels that inherit from a GameViewModel class which in-turn inherits from BaseViewModel.

The GameViewModel class implements publically accessible properties that are used in a game. Below are a couple of example fields that are shown on a game screen:



For example, we have a RoundTimeLeft property that shows how much time you have left in a round. The property is of type TimeSpan and it uses a private backing field. When you set the property, we use a method of the NotificationObject class to notify the View layer that a ViewModel property has been updated.

```
private TimeSpan _roundTimeLeft;
public TimeSpan RoundTimeLeft
{
    get { return _roundTimeLeft; }
    private set
    {
        _roundTimeLeft = value;
        RaisePropertyChanged(() => RoundTimeLeft);
    }
}
```

This is especially useful in situations where you want the View to refresh multiple properties when you update a single field/property. Also, as a performance improvement for advanced applications, it is very common to check if the value has been changed before notifying the view that your property has changed. Below is an example of the HintItem property and Hint property that are used in the ViewModel. The Hint property is the symbol that is shown in the center, and we want to update that text anytime we store a new HintItem in the ViewModel. This is done by letting the View know that the Hint property has been updated:

```
private PeriodicItem _hintItem;
public string Hint
```

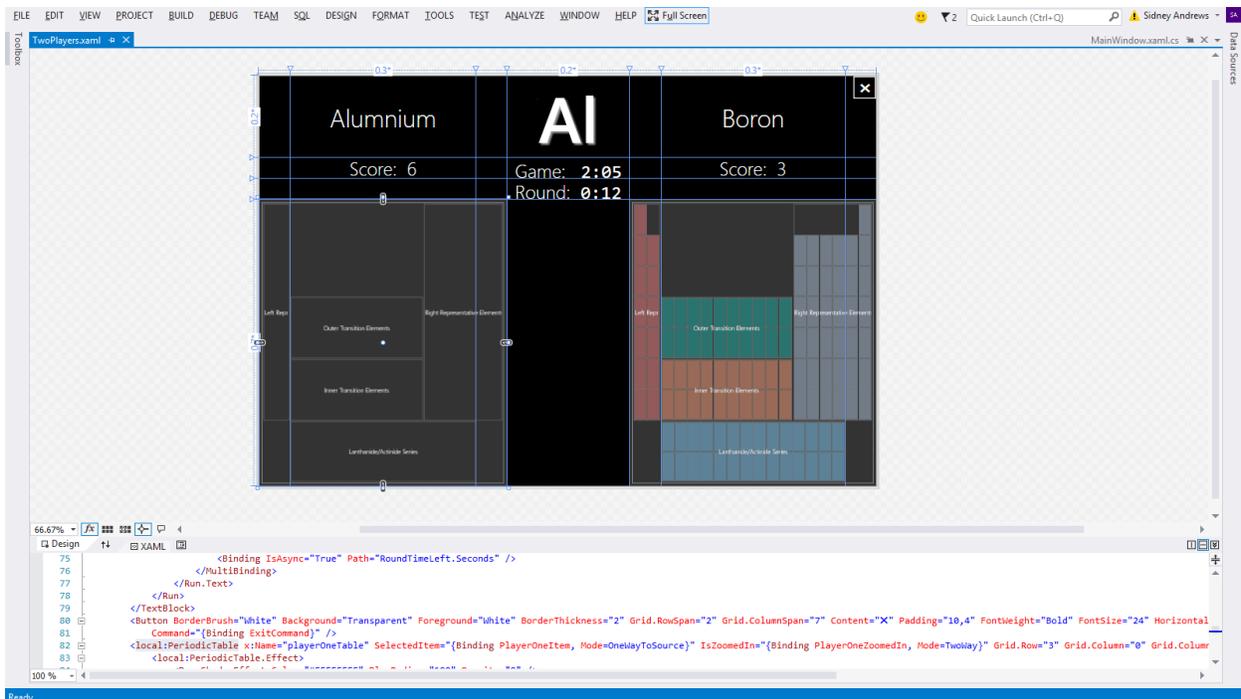
```

    {
        get
        {
            return this.HintItem != null ? this.HintItem.Abbreviation : string.Empty;
        }
    }

    public PeriodicItem HintItem
    {
        get { return _hintItem; }
        private set
        {
            _hintItem = value;
            RaisePropertyChanged(() => Hint);
            RaisePropertyChanged(() => HintItem);
        }
    }
}

```

The Model-View-ViewModel pattern is very powerful and allows testability and expanded code re-use when working with an application. The pattern is also applicable whether you are working on a line-of-business application or a touch application. The GameViewModel class uses a timer and a loop to handle the execution of the game. Both OnePlayerViewModel and TwoPlayersViewModel inherit from the GameViewModel and add specific logic for each type of game. The application also has a DesignGameViewModel that has a set of static properties so that we can see how the game will look at design time without having to run the application:



## Tips & Tricks for building immersive applications in WPF

There are a couple of XAML tricks that are used throughout this application to make it visually appealing and touch friendly. Some are very common, but there are a couple worth highlighting as they use some of the best features of WPF and XAML.

First, the PeriodicTable itself is a WPF UserControl. This allows maximum code re-use as the control can simply be placed on any WPF Window. Within the control, Dependency Properties are used so that you can set features of the control and expose those features externally for data-binding. For example, the PeriodicTable has two states. ZoomedOut is when you see the entire table:

The image shows a zoomed-out view of the periodic table. The elements are grouped into several categories, each highlighted with a different color:

- Left Representative Elements:** Highlighted in red, located on the far left.
- Outer Transition Elements:** Highlighted in teal, located in the middle-left section.
- Inner Transition Elements:** Highlighted in brown, located in the middle section.
- Right Representative Elements:** Highlighted in grey, located on the far right.
- Lanthanide/Actinide Series:** Highlighted in blue, located at the bottom.

ZoomedIn is when you see the detailed list. When clicking on a Periodic Group from the ZoomedOut view, the game jumps to that group on the ZoomedIn list. There is also a button in the bottom-right corner to zoom back out:

The image shows a zoomed-in view of the periodic table, displaying a detailed list of elements. The elements are arranged in a grid, with their symbols, names, and atomic numbers visible. The elements shown are:

Right Representative Elements							
He	N	Ne	P	Ar	As	Kr	Sb
Helium 2	Nitrogen 7	Neon 10	Phosphorus 15	Argon 18	Arsenic 33	Krypton 36	Antimony 51
B	O	Al	S	Ga	Se	In	Te
Boron 5	Oxygen 8	Aluminum 13	Sulfur 16	Gallium 31	Selenium 34	Indium 49	Tellurium 52
C	F	Si	Cl	Ge	Br	Sn	I
Carbon 6	Fluorine 9	Silicon 14	Chlorine 17	Germanium 32	Bromine 35	Tin 50	Iodine 53

At the bottom right, there is a button with a minus sign, used to zoom back out.

To implement this, there are two list views representing each of the “Views.” A dependency property is created that will expose a property that anybody can set. A PropertyChanged event handler is then created so that the control can respond to changes from both code and data-bindings all in one location:

```
public static readonly DependencyProperty IsZoomedInProperty =
DependencyProperty.Register(
    "IsZoomedIn", typeof(bool), typeof(PeriodicTable),
    new PropertyMetadata(false, ZoomedInChanged)
);

public bool IsZoomedIn
{
    get { return (bool)GetValue(IsZoomedInProperty); }
    set { SetValue(IsZoomedInProperty, value); }
}

public void SetZoom(bool isZoomedIn)
{
    if (IsZoomedIn)
    {
        FirstContainer.Visibility = Visibility.Collapsed;
        SecondContainer.Visibility = Visibility.Visible;
    }
    else
    {
        FirstContainer.Visibility = Visibility.Visible;
        SecondContainer.Visibility = Visibility.Collapsed;
    }
}
```

This dependency property is used in the TwoPlayerView so that we can bind the Second Player’s zoomed in state to a Boolean in the ViewModel called PlayerTwoZoomedIn:

```
<local:PeriodicTable x:Name="playerTwoTable" IsZoomedIn="{Binding PlayerTwoZoomedIn,
Mode=TwoWay}"></local:PeriodicTable>
```

This implementation loses the flexibility to tie custom features from the control to anything in the ViewModel. In our application, we need to set PlayerTwoZoomedIn (and PlayerOneZoomedIn) to false when a round or the game is reset.

XAML is also heavily used to store the data in this application. While a database or a text file could be created, it seemed to be much more readable to store the Periodic table’s data as XAML. Since XAML is just an XML representation of CLR objects, we could create model classes and corresponding XAML elements. We can then store this in a XAML resource dictionary and load it as data at runtime (or design time if you wish).

For example, we have a class for PeriodicItems that has a very simple definition and is represented by even simpler XAML:

```
public class PeriodicItem
{
    public string Title { get; set; }

    public string Abbreviation { get; set; }
}
```

```

    public int Number { get; set; }
}

<local:PeriodicItem Abbreviation="Sc" Title="Scandium" Number="21" />
<local:PeriodicItem Abbreviation="Ti" Title="Titanium" Number="22" />

```

This made defining the Periodic table easy and readable. You can find all of the Periodic elements used in the application in the **PeriodicTableDataSource.xaml** file located in the Data folder. Here is an example of a Periodic Group defined in that file.

```

<local:PeriodicGroup Key="Outer Transition Elements">
  <local:PeriodicGroup.Items>
    <local:PeriodicItem Abbreviation="Ni" Title="Nickel" Number="28" />
    <local:PeriodicItem Abbreviation="Cu" Title="Copper" Number="29" />
    <local:PeriodicItem Abbreviation="Zn" Title="Zinc" Number="30" />
    <local:PeriodicItem Abbreviation="Y" Title="Yttrium" Number="39" />
  </local:PeriodicGroup.Items>
</local:PeriodicGroup>

```

Because of this, the Periodic data is dynamic and can be modified by simply updating the .xaml file. You can also use the same data in both design and runtime Views since it's compiled and available as a resource for XAML.

## Summary

Building an application that supports a large amount of data and advanced touch scenarios is definitely possible in the Windows 8 desktop environment. XAML is a powerful markup language that allows you to not only define dynamic views, but also model your data in a common format that is very easy to read, understand, and parse. You can build touch applications today using the mature WPF platform.

## Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: <http://www.intel.com/design/literature.htm>

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark\* and MobileMark\*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

Copyright © 2013 Intel Corporation. All rights reserved.

\*Other names and brands may be claimed as the property of others.