TMS MultiTouch SDK

Multitouch background

The first multitouch experience offered by Microsoft was the Microsoft Surface introduced in May 2007. The Microsoft Surface consisted of quite sophisticated hardware made up of a projector and 5 infrared cameras. The software consisted of an SDK and drivers for Windows Vista. The SDK targetted the WPF framework in the first place. The hardware was very expensive and the experience to create apps with WPF and the SDK was daunting. With Windows 7, Microsoft added built-in support for multitouch and we saw the introduction of low cost hardware like the Dell ST220T multitouch monitor or the HP TouchSmart machine. This means that at this point, the only remaining obstacle to create multitouch applications is writing the software. With Microsoft technologies, this mainly still means going the WPF route and this can be cumbersome, time-intensive and complex at times.

Multitouch basics

Making an application multitouch-aware typically means:

1) Handling multitouch events
2) Manipulating objects synchronously with the multitouch events
3) Optionally handling inertia

In the next 3 paragraphs, each of these steps will be explained.

Handling multitouch events

From Windows 7, a new message is defined: WM_TOUCH. This new window message is the main starting point to begin handling multitouch. The definition of this message is:

WM_TOUCH: Notifies the window when one or more touch points, such as a finger or pen, touches a touch-sensitive digitizer surface.

The WM_TOUCH message returns following parameters:

Low-order word of the wParam parameter contains the number of touch points associated with this message. The lParam parameter contains a handle that can be used with the new call GetTouchInputInfo(). With this method, detailed information about the touch points can be retrieved.

The definition of the new GetTouchInputInfo() API function is:

```
BOOL WINAPI GetTouchInputInfo(
  __in   HTOUCHINPUT hTouchInput,
  __in   UINT cInputs,
  __out  PTOUCHINPUT pInputs,
  __in   int cbSize
);
```

with: hTouchInput, the handle that was passed in the lParam parameter of the WM_TOUCH message. cInputs contains the number of structures passed via the pInputs parameter. pInputs is a pointer to an array of _TOUCHINPUT structures. Note that when the

GetTouchInputInfo() was called, this should be followed by a call to CloseTouchInputHandle(). Fortunately, from Delphi 2010, this new message identifier, the new API method and structures are all declared. So, in Delphi, if we'd want to handle multitouch input, this is typically done in following way:

```
TMyMultiTouchControl = class(TCustomControl)
private
  procedure WMTouch(var Message: TMessage); message WM_TOUCH;
end;

procedure TMultiTouchRegion.WMTouch(var Message: TMessage);
var
  TouchInput: array of TTouchInput;
  i: integer;
begin
  SetLength(TouchInput, message.WParam);
  GetTouchInputInfo(Message.LParam, Message.WParam, @TouchInput[0],
sizeof(TTouchInput));

  for i := 0 to message.wParam - 1 do

outputdebugstring(PChar(IntToStr(TouchInput[i].x)+':'+IntToStr(TouchInput[i].y))
);

  CloseTouchInputHandle((Message.lParam)

  inherited;
end;
```

Manipulating objects

To handle the synchronous manipulation of objects with movement of touch points, the Windows 7 SDK introduces the IManipulationProcessor COM interface. This interface is declared in the Delphi unit Manipulations.pas. For each object that we want to have manipulated, it is required to create an instance of the IManipulationProcessor and connect it with a Delphi object that implements the _IManipulationEvents interface. The task of the ManipulationProcessor is to convert the incoming touch point information to transformations in terms of translation/rotation/scaling of objects.

The IManipulationEvents interface is defined as:

```
  _IManipulationEvents = interface(IUnknown)
    [SID__IManipulationEvents]
    function ManipulationStarted(x: Single; y: Single): HRESULT; stdcall;
    function ManipulationDelta(x: Single; y: Single; translationDeltaX: Single;
      translationDeltaY: Single; scaleDelta: Single; expansionDelta: Single;
      rotationDelta: Single; cumulativeTranslationX: Single;
      cumulativeTranslationY: Single; cumulativeScale: Single;
      cumulativeExpansion: Single; cumulativeRotation: Single): HRESULT;
stdcall;
    function ManipulationCompleted(x: Single; y: Single;
      cumulativeTranslationX: Single; cumulativeTranslationY: Single;
```

```
    cumulativeScale: Single; cumulativeExpansion: Single;
    cumulativeRotation: Single): HRESULT; stdcall;
end;
```

So, basically, to manipulate objects, for each touchpoint, identify the object under the touchpoint, call the IManipulationProcessor's ProcessDown, ProcessMove, ProcessUp methods and then the IManipulationProcessor will call the object's _IManipulationEvents interface methods to update the transformation matrix we can associate with the object to set its translation, rotation and scale.

Handling inertia

To create natural user interfaces, handling inertia is also a critical part. The definition of inertia is: the resistance of any physical object to a change in its state of motion or rest, or the tendency of an object to resist any change in its motion. In terms of manipulating objects on the screen, that means that when moving an object with a given velocity on the screen, moving the finger will not immediately cause the object to stop but instead slowly decrease speed due to so called resistance of the surface till it eventually stops. Technically, handling inertia works in a similar way as manipulation. Each object creates an instance of the IInertiaProcessor COM interface and connects this interface with the object's IManipulationEvents interface. In the ManipulationCompleted() method, we retrieve the parameters of the touchpoint movement, i.e. the velocity in X and Y direction and the angle of velocity and initialize the IInertiaProcessor with these parameters. It is then the IInertiaProcessor that will be responsible for converting these velocities in further manipulations of the object after the actual manipulation with touch ended. Via a timer, the further inertia based manipulation is requested with: InertiaProcessor.Process(Completed); When the inertia effect is finished, it will set the parameter Completed to true, indicating no further processing of the inertia effect is required.

Entering the TMS MultiTouch SDK

You might have come to the conclusion from the above technical description of multitouch handling that this is far from trivial. At TMS, we thought it would be way more convenient to encapsulate all these complexities in a VCL component: TMultiTouchRegion that handles all this technical stuff in the background for you and just expose the capabilities by a set of properties and events. In addition to the TMultiTouchRegion that is the core component for handling multi-touch, the TMS MultiTouch SDK comes with over 30 additional touch supporting controls as typical touch based interfaces require much more possibilities than just manipulating objects with multi-touch.

TMultiTouchRegion component architecture

The TMultiTouchRegion is built-up of 4 layers. The background or the region is the bottom layer. Note that the background can also be moved, scaled, rotated. This means that all items that will be placed on the background will be transformed along with the transformation of the background layer. Note that this also means that the background can be much larger than the actual control size and multi-touch interaction will allow to scale or perform movements on this region. Immediately on top of this background layer is a layer of control items. Control items are items that cannot be moved, sized or rotated. The items have a fixed position and the only interaction with this item type is a click, exposed by the event OnControlItemClick. Control items in this layer are added via the collection TMultiTouchRegion.ControlItems. Typical purpose of background control items is for example a button to exit the application, rotate the screen etc.. The next layer consists of regular items. Regular items can be manipulated via multi-touch to scale, rotate or translate the items. The regular items, added via the MultiTouchRegion.Items collection, will as such always be displayed on top of the background of background control items.

The regular items are a collection of items of the type TMultiTouchItem. The published properties of TMultiTouchItem are:

```
TMultiTouchItem
published
  CanMoveHorizontal: boolean;
  CanMoveVertical: boolean;
  CanRotate: boolean;
  CanScale: boolean;
  DetailItem: TDisplayInfo;
  EnableFlip: boolean;
```

```
  ItemHeight: integer;
  ItemWidth: integer;
  ItemX: integer;
  ItemY: integer;
  LinkedItem: TMultiTouchItem;
  MainItem: TDisplayInfo;
  Resizable: boolean;
  ResizeHandleSize: integer;
  StackIndex: integer;
end;
```

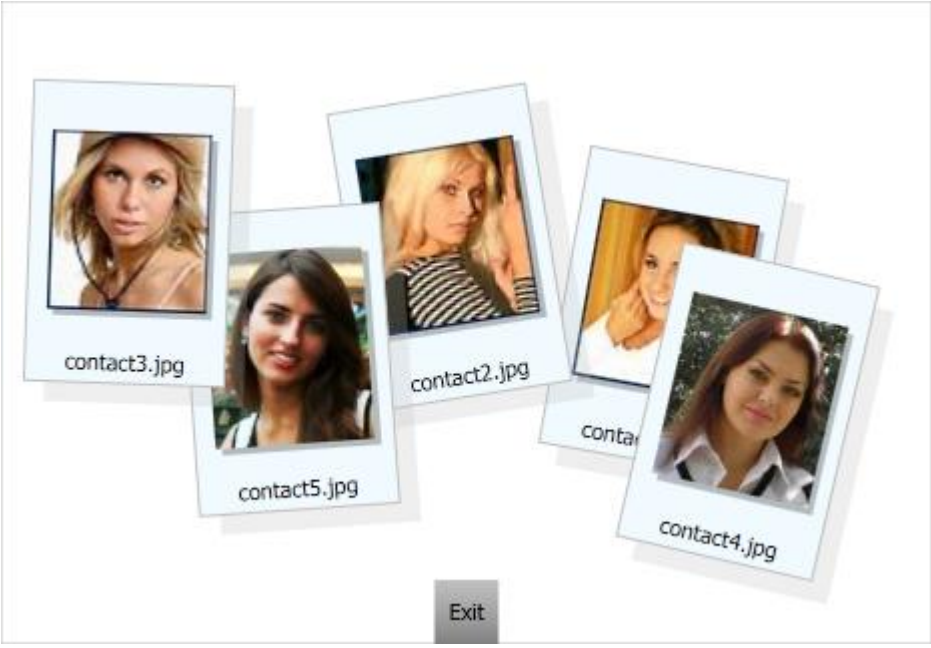
With just simple boolean properties (CanMoveHorizontal, CanMoveVertical, CanRotate, CanScale), it can be controlled what type of movements of items are possible, if the item can be rotated or scaled. An item itself has a foreground (MainItem) and a background (DetailItem) side. Having a main side and detail side allows effects just as clicking on an item to make it flip between main side & detail side. The item main or detail side can display an image, text or the combination of text and images. An item can be linked to another item via the property LinkedItem. When two items are linked, this means that translations, rotations, scaling on one item will have the same effect on the second item. Note that it is possible to create a chain of linked items.

On top of the regular items is a last layer of static control items. Similar to background control items, these items cannot be moved, scaled, rotated. The only interaction is also clicking on an item.

Getting started with the TMultiTouchRegion

To show how easy and fast immersive multitouch user interfaces can be created with TMultiTouchRegion, this sample code snippet will add all JPEG files from a folder and add these to the TMultiTouchRegion and add a static control item to leave the application when clicked:

```
begin
  // sets the default size of items in the multitouch region
  MultiTouchRegion1.DefaultItem.ItemWidth := 100;
  MultiTouchRegion1.DefaultItem.ItemHeight := 150;
  // Add all image files in the specified folder to be loaded by a thread by the
component
  MultiTouchRegion1.AddFileLocationsFromFolder('\My Pictures\*.jpg');
  // Add a control item of size 32x32 pixels in the top left corner of the
control
  MultiTouchRegion1.ControlItems.Add.Text := 'Exit';
  MultiTouchRegion1.ControlItems[0].Tag := 1;
  MultiTouchRegion1.ControlItems[0].Top := 0;
  MultiTouchRegion1.ControlItems[0].Left := 0;
  MultiTouchRegion1.ControlItems[0].Width := 32;
  MultiTouchRegion1.ControlItems[0].Height := 32;
end;

// handle the closing from the static control item click
procedure TForm1.MultiTouchRegion1ControlItemClick(Sender: TObject;
  ControlItem: TMultiTouchCustomItem);
begin
  if ControlItem.Tag = 1 then
    Close;
end;
```



The AddFileLocationsFromFolder will loop through the matching JPG files in the folder and add these to the MultiTouchRegion.Items collection. Note that it is a background thread in the TMultiTouchRegion component that will perform the actual loading of the images as the call to AddFileLocationsFromFolder will immediately return after all file names have been retrieved.

It is also possible to add items one by one programmatically. This can be done either by setting the item's FileName property and the components internal background thread will perform the actual loading:

```
MultiTouchRegion1.Items.Add.MainItem.FileName := myfilename;
```

or the image can be loaded under application code control:

```
MultiTouchRegion1.Items.Add.MainItem.Picture.LoadFromFile(myfilename);
```

TMultiTouchRegion display modes

As each regular item can be moved, scaled, rotated, the TMultiTouchRegion keeps track of these transformations via transform matrices. As each item can have a background rectangle and the actual content rectangle, two transforms are always associated with an item:

```
MultiTouchRegion.Items[index].MainItem.BackgroundTransform: TD2DMatrix3x2F;
MultiTouchRegion.Items[index].MainItem.ContentTransform: TD2DMatrix3x2F;
```

The background of an item is an optional rectangular area that is displayed as placeholder for the content (image/text) and has its own transfer. When items are manipulated via multi-touch, it is these transform matrices that will be updated. The transform matrices are read/write properties. This means that the position of an item can also be set under

programmatic control via these matrices and the TMultiTouchRegion itself will also initialize these matrices dependent on the chosen display mode.

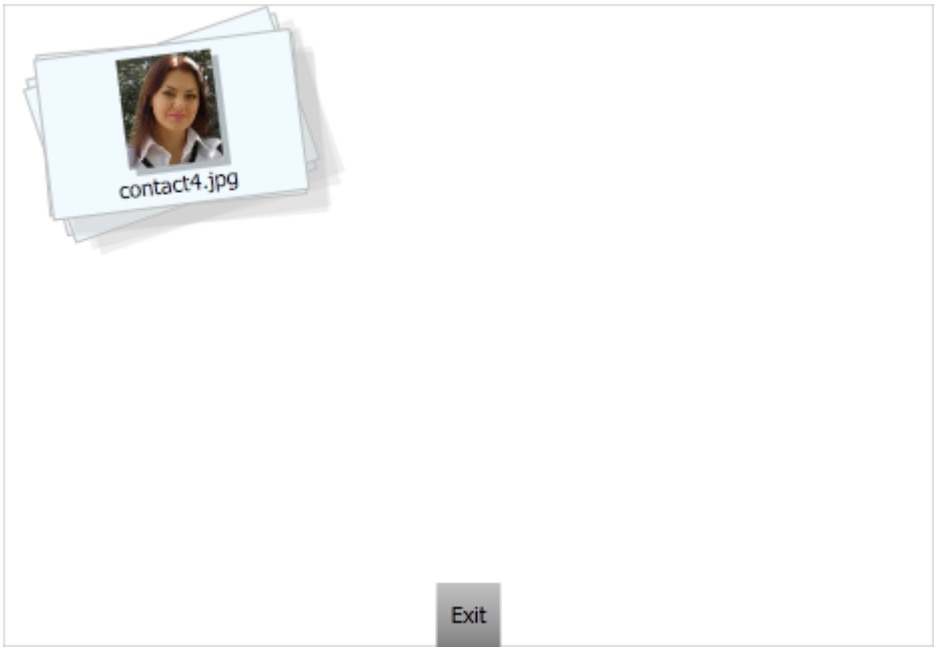
```
MultiTouchRegion.DisplayMode: TMultiTouchDisplayMode
```

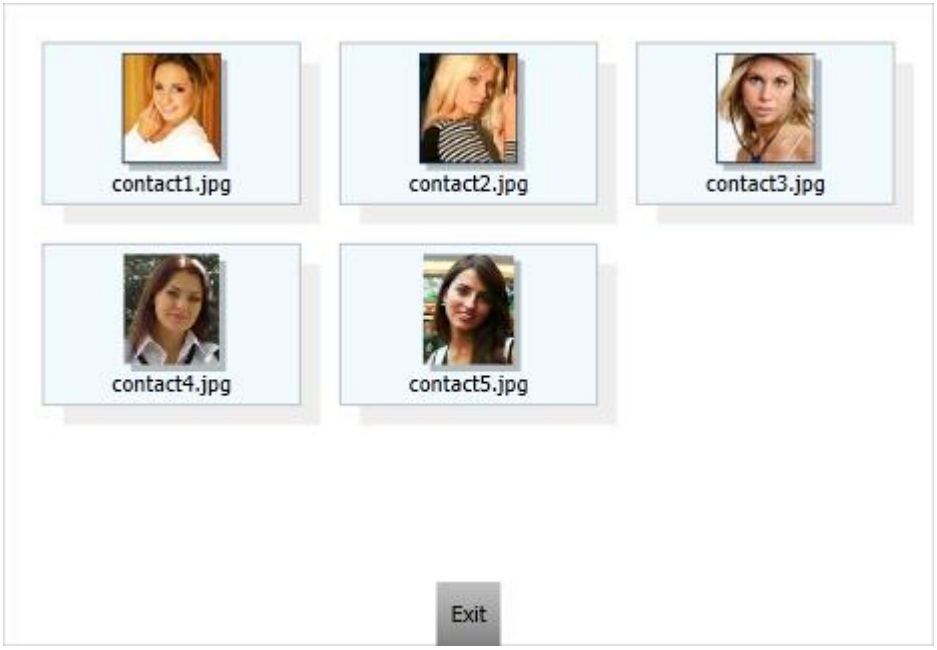
TMultiTouchDisplayMode is defined as:

- dmRandom: matrices of items are initialized in a random way by the MultiTouchRegion as items are added
- dmGrid: matrices are setup in such a way that items are displayed as a grid using the Cols/Rows properties
- dmCascade: matrices are setup in such a way that items are displayed in cascade style, using HorizontalSpacing,VerticalSpacing properties
- dmPosition: matrices are setup to display items in a not rotated and not scaled way at position Item.ItemX, Item.ItemY.
- dmMatrices: items will be displayed with the matrices configured as is.
- dmStacked: items are displayed in grid mode organized in stacks. All items with the same Item.StackIndex property value will be displayed on top of each other to form a stack.

It is the DisplayMode = dmMatrices that can be used to persist the last location & manipulation of items. The TMultiTouchRegion provides the built-in method MultiTouchRegion.SaveToFile() / MultiTouchRegion.LoadFromFile() that will save & restore the full item's setup and matrices.

This sample code snippet loads the images as a stack. With the property MultiTouchRegion.OpenStackMode set to dmGrid, this causes that when the stack is clicked, the TMultiTouchRegion will automatically open the stack with animation to show the images in dmGrid mode:





Item controls

Sometimes it is desirable to have controls associated with an item. This could be a button to delete an item, a button to open a software keyboard on screen to enter text for an item etc... It is expected that such a button control will move, rotate and scale along with an item as it is manipulated. To make it easy to add such controls, the item exposes a Controls collection on Item.MainItem and Item.DetailItem level. A control has a normal state and a down state and the appearance of these 2 states can be set via:

```
MultiTouchRegion.Items[index].MainItem.Controls[controlindx].Fill
```

and

```
MultiTouchRegion.Items[index].MainItem.Controls[controlindx].FillDown
```

The Fill class can for example be used to specify a PNG image used for the button.

The position of the control relative to the item is set with:

```
MultiTouchRegion.Items[index].MainItem.Controls[controlindx].Position
```

Finally, handling a click of such a control is handled via the event TMultiTouchRegion.OnItemControlItemClick(). This event returns an instance of the ControlItem in the Item.MainItem.Controls collection and an instance of the item itself.

This sample code snippet shows how to add a Delete control in the top right corner of an item that is only visible when the item is selected. The OnItemControlItemClick event will then delete the item:

```
procedure TForm1.FormCreate(Sender: TObject);
```

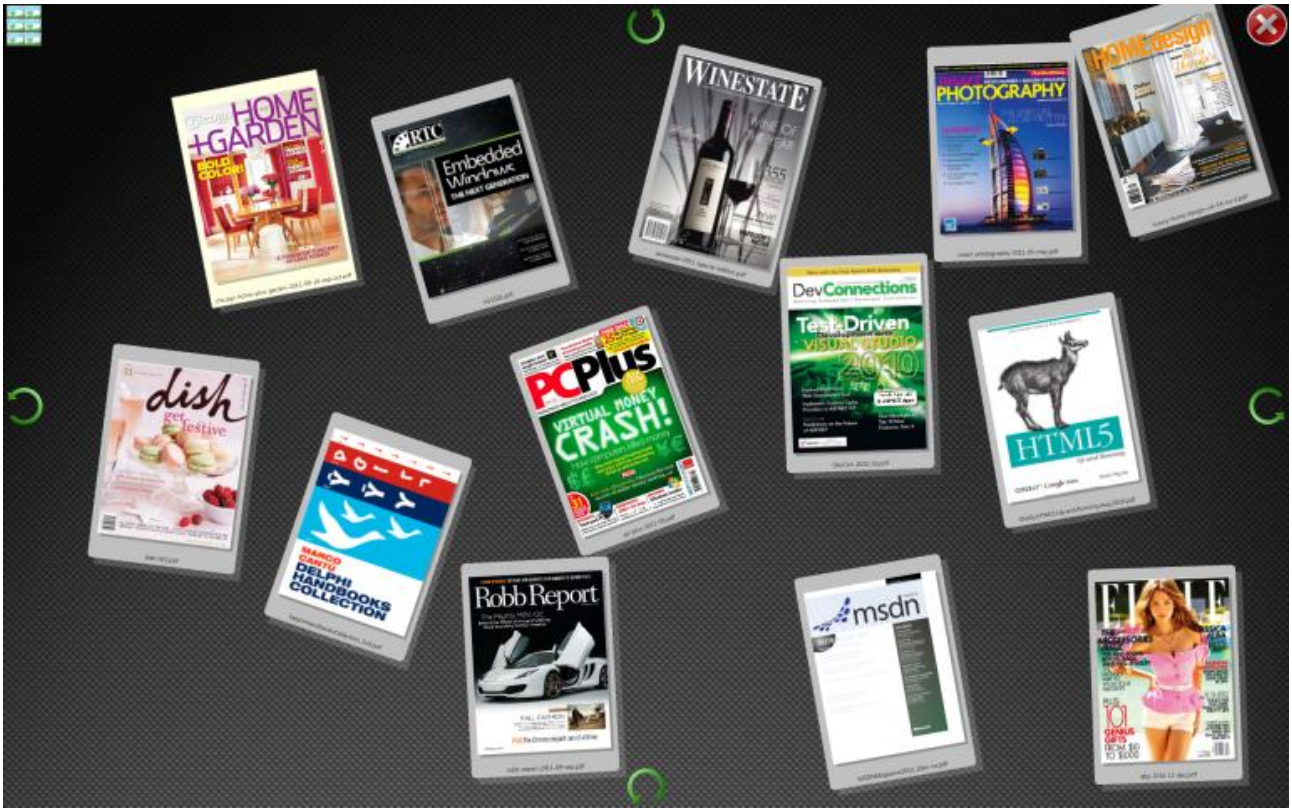
```
var
  ctrl: TMultiTouchCustomItem;
begin
  MultiTouchRegion1.Items.Add;

  ctrl := MultiTouchRegion1.Items[0].MainItem.Controls.Add;
  ctrl.Location := ilTopRight;
  ctrl.Width  := 16;
  ctrl.Height := 16;
  ctrl.Fill.Picture.LoadFromFile("cancelpic.png");
  ctrl.Visible := false;
end;

procedure TForm1.MultiTouchRegion1ItemSelectedChanged(Sender: TObject;
  Item: TMultiTouchItem);
begin
  Item.MainItem.Controls[0].Visible := Item.Selected;
end;

procedure TForm1.MultiTouchRegion1ItemControlItemClick(Sender: TObject;
  ControlItem: TMultiTouchCustomItem; Item: TMultiTouchItem);
begin
  Item.Free;
end;
```

Visualizers

By default, an item in the TMultiTouchRegion can display an image in various file formats: BMP,JPEG,PNG,GIF or just text or a combination of image and text. To make the component flexible to show other types of files or formats, a visualizer virtual class is provided. When a visualizer is associated with the TMultiTouchRegion, it is the visualizer's protected virtual method LoadImage that will get called whenever the TMultiTouchRegion component needs it to draw a visual representation of the item. The LoadImage() override is supposed to return this via a TD2DPicture, a Direct2D image. It is very easy to create a custom visualizer. The minimum implementation is a class that descends from and implements LoadImage().

```
  TMyVisualizer = class(TMultiTouchRegionVisualizer)
  protected
    procedure LoadImage(APicture: TD2DPicture; D2DItem: TDisplayInfo; AThread:
TMultiTouchThread);
  end;
```

The TMultiTouchRegion comes default with 3 visualizers. The built-in visualizer for the most common image file formats, a TMultiTouchRegionPreviewVisualizer to use the shell preview handler to display the shell preview image for a given file and also a TMultiTouchRegionPDFVisualizer that is able to render pages of a PDF file. Below is a screenshot of such TMultiTouchRegion where the PDFVisualizer renders the cover page of a number of magazines. Note also the 4 control items (green circular arrows) at each side of control from where screen rotation is performed and the control item in the top right corner to exit the application and the control item in the top left corner to line up the items in grid mode.



Summary

With Windows 7, Microsoft provides the software infrastructure to create multitouch applications. The provided APIs are unfortunatately still quite complex and daunting to start with. With the TMS MultiTouch SDK, TMS software encapsulates all these complexities in a TMultiTouchRegion component and makes it accessible and very fast to create immersive touch applications. The TMS MultiTouch SDK also comprises the full range of TMS touch-centric smooth controls with which graphically attractive full-blown point of sales, multimedia, educative software can be created. On top of that, TMS software also offers flexible multi-touch hardware solutions, starting with a 32" 6 point multitouch table or wall model till solutions up to 60". Or, you can also outsource your entire multi-touch project as TMS

software offers the services to create this for you. Contact us for more information and details.
Website page: http://www.tmssoftware.com/site/multitouchsdk.asp