**Contoso Jobs**

**Lab 1. Adding Multi-User Multi-Touch Drag and Drop**

With a large screen like Surface Hub multiple people can touch the screen at the same time. This means any functionality you add should be available for multiple people, at the same time, on the same screen.

A good touch application should apply the principle of direct manipulation. When an item needs to be moved between columns you should be able to use your finger to pick up the item and move it.

In this lab you will learn how to modify a simple UWP application to support drag and drop for multiple people at the same time. The App will enable the Jobs to dragged and dropped within the app, between the columns to represent the status of the Job.

1. Open the existing application.   
   Find the folder named **1. Start Contoso Jobs Drag Drop**In this folder open the **Contoso Jobs.sln** solution file in **Visual Studio 2015**
2. In **Solution Explorer** open the **ViewModels** folder and you will see a code file named **JobViewModel.cs**. Open this file. In this **JobViewModel** class you will add two methods to move a job between columns.  
   The first method you add will start to move a Job, it will change the status of the Job to **Moving**.

Please note this method will compile because the **JobStatus** enum type already contains the value 'Moving' (**Models | JobStatus.cs**).

internal void MoveJob(Job job)

{

if (null != job)

{

SaveJobs();

switch (job.Status)

{

case JobStatus.Backlog:

job.Status = JobStatus.Moving;

OnPropertyChanged("Backlog");

break;

case JobStatus.WIP:

job.Status = JobStatus.Moving;

OnPropertyChanged("WIP");

break;

case JobStatus.Done:

job.Status = JobStatus.Moving;

OnPropertyChanged("Done");

break;

default:

break;

}

}

}

1. Add a method to move a job directly to a specific status. This can be used for direct manipulation of a Job.

internal void MoveJob(Job job, JobStatus status)

{

if (job != null)

{

if (status == JobStatus.Backlog)

{

job.Status = JobStatus.Backlog;

OnPropertyChanged("Backlog");

}

else if (status == JobStatus.WIP)

{

job.Status = JobStatus.WIP;

OnPropertyChanged("WIP");

}

else if (status == JobStatus.Done)

{

job.Status = JobStatus.Done;

OnPropertyChanged("Done");

}

SaveJobs();

}

}

1. In Solution Explorer find the Common folder and create a new class file named ManipulationInputProcessor.cs
2. Change the class to be internal so that other code in the project can access this class

internal class ManipulationInputProcessor

{

}

1. Edit the using statements at the top of the class file to make it easier to get intellisense for the code you will add in the following steps.

using Contoso\_Jobs.Models;

using Contoso\_Jobs.Views;

using Windows.UI;

using Windows.UI.Input;

using Windows.UI.Xaml;

using Windows.UI.Xaml.Controls;

using Windows.UI.Xaml.Input;

using Windows.UI.Xaml.Media;

1. Add member fields to the class to hold the GestureRecognizer object and UIElements for the movement of Jobs between columns

private GestureRecognizer recognizer;

private UIElement element;

private UIElement reference;

1. Add a constructor method to the class that takes a GestureRecognizer and two UIElements as parameters. This constructor will store the parameters in the local member fields and then set up event handlers for the Pointer and Manipulation events

public ManipulationInputProcessor(GestureRecognizer gestureRecognizer,

UIElement target, UIElement referenceFrame)

{

recognizer = gestureRecognizer;

element = target;

reference = referenceFrame;

// The GestureSettings property dictates what manipulation events the

// Gesture Recognizer will listen to. This will set it to a limited

// subset of these events.

recognizer.GestureSettings = GenerateDefaultSettings();

// Set up pointer event handlers.

// These receive input events that are used by the gesture recognizer.

element.PointerPressed += OnPointerPressed;

element.PointerMoved += OnPointerMoved;

element.PointerReleased += OnPointerReleased;

element.PointerCanceled += OnPointerCanceled;

// Set up event handlers to respond to gesture recognizer output

recognizer.ManipulationStarted += OnManipulationStarted;

recognizer.ManipulationUpdated += OnManipulationUpdated;

recognizer.ManipulationCompleted += OnManipulationCompleted;

recognizer.ManipulationInertiaStarting += OnManipulationInertiaStarting;

}

1. Add a method to setup the Gestures that need to be recognized by this App. The TranslateX and TranslateY gestures will enable the App to support direct manipulation of dragging the Jobs around.

// Return the default GestureSettings for this sample

GestureSettings GenerateDefaultSettings()

{

return GestureSettings.ManipulationTranslateX

| GestureSettings.ManipulationTranslateY

;

}

1. Add an event handler for the PointerPressed event to route the event to the GestureRecognizer

// Route the pointer pressed event to the gesture recognizer.

// The points are in the reference frame of the canvas that contains the rectangle element.

public void OnPointerPressed(object sender, PointerRoutedEventArgs args)

{

// Set the pointer capture to the element being interacted with so that only it

// will fire pointer-related events

element.CapturePointer(args.Pointer);

// Feed the current point into the gesture recognizer as a down event

recognizer.ProcessDownEvent(args.GetCurrentPoint(reference));

}

1. Add method for PointerMoved to allow the recognizer to process the event. This pattern is used for the pointer events to enable the GestureRecognizer to do it’s job.

// Route the pointer moved event to the gesture recognizer.

// The points are in the reference frame of the canvas that contains the rectangle element.

void OnPointerMoved(object sender, PointerRoutedEventArgs args)

{

// Feed the set of points into the gesture recognizer as a move event

recognizer.ProcessMoveEvents(args.GetIntermediatePoints(reference));

}

1. Add method for PointerReleased, forwarding the event details to the recognizer

// Route the pointer released event to the gesture recognizer.

// The points are in the reference frame of the canvas that contains the rectangle element.

void OnPointerReleased(object sender, PointerRoutedEventArgs args)

{

// Feed the current point into the gesture recognizer as an up event

recognizer.ProcessUpEvent(args.GetCurrentPoint(reference));

// Release the pointer

element.ReleasePointerCapture(args.Pointer);

if (element.PointerCaptures != null

&& element.PointerCaptures.Count < 1)

{

recognizer.CompleteGesture();

}

}

1. Add method for PointerCanceled, again notifying the GestureRecognizer, and this time also releasing the capture on the Pointer.

// Route the pointer canceled event to the gesture recognizer.

// The points are in the reference frame of the canvas that contains the rectangle element.

void OnPointerCanceled(object sender, PointerRoutedEventArgs args)

{

recognizer.CompleteGesture();

element.ReleasePointerCapture(args.Pointer);

}

1. Add a method to handle the ManipulationStarted event to change the highlight color of the Job, so it is clear to the person that they are now manipulating the object.

// When a manipulation begins, change the color of the object to reflect

// that a manipulation is in progress

void OnManipulationStarted(object sender, ManipulationStartedEventArgs e)

{

JobControl jc = element as JobControl;

jc.Highlight = new SolidColorBrush(Colors.Red);

}

1. Add a method for the ManipulationUpdated event to translate the position of the Job on the canvas.

// Process the change resulting from a manipulation

void OnManipulationUpdated(object sender, ManipulationUpdatedEventArgs e)

{

Transform t = element.RenderTransform;

if (t != null

&& t is CompositeTransform)

{

CompositeTransform ct = t as CompositeTransform;

ct.TranslateX += e.Delta.Translation.X;

ct.TranslateY += e.Delta.Translation.Y;

}

else if (t is TranslateTransform)

{

TranslateTransform trans = t as TranslateTransform;

trans.X += e.Delta.Translation.X;

trans.Y += e.Delta.Translation.Y;

}

}

1. In order to reflect that Inertia has taken over from the manipulation add a method to handle the intertia starting event. In this example you will simply change the highlight color.

// When a manipulation that's a result of inertia begins, change the color of the

// the object to reflect that inertia has taken over

void OnManipulationInertiaStarting(object sender,

ManipulationInertiaStartingEventArgs e)

{

JobControl jc = element as JobControl;

jc.Highlight = new SolidColorBrush(Colors.Green);

}

1. Add a method to handle the manipulation ending that will determine the column which the Job is currently over and move the job to that column.

// When a manipulation has finished, reset the color of the object

void OnManipulationCompleted(object sender, ManipulationCompletedEventArgs e)

{

if (element is JobControl)

{

JobControl jc = element as JobControl;

if (jc.DataContext is Job)

{

Job j = jc.DataContext as Job;

jc.Highlight = new SolidColorBrush(Colors.DarkGray);

//need to determine which column it is over

Grid g = reference as Grid;

double width = g.ActualWidth;

double columnWidth = width / 3;

if (e.Position.X < columnWidth)

{

Jobs.jobsViewModel.MoveJob(j, JobStatus.Backlog);

}

else if (e.Position.X < columnWidth \* 2)

{

Jobs.jobsViewModel.MoveJob(j, JobStatus.WIP);

}

else

{

Jobs.jobsViewModel.MoveJob(j, JobStatus.Done);

}

g.Children.Remove(jc);

}

}

}

You have now added the code needed to handle manipulations of the Job objects.

When a touch is applied to a Job control a copy of that Job control is created and added to the Jobs grid

The final task is to forward the **Pointer** (or touch) events from the Job object to the **ManipulationInputProcessor**.

1. Open the **Jobs.xaml.cs** code file (in the **Views** Folder) and add a method to handle the **PointerPressed** event.

private void JobControl\_PointerPressed(object sender, PointerRoutedEventArgs e)

{

if (sender is JobControl)

{

JobControl selectedJobControl = sender as JobControl;

Job j = selectedJobControl.DataContext as Job;

if (j == null)

{

return;

}

JobControl dragJobControl = new JobControl();

dragJobControl.DataContext = j;

//set the visual of the new dragging job control

dragJobControl.Fill = new SolidColorBrush(Colors.LightGray);

Grid.SetColumnSpan(dragJobControl, 3);

dragJobControl.Height = selectedJobControl.ActualHeight;

dragJobControl.Width = selectedJobControl.ActualWidth;

// position the new dragging job control on the grid in the same position

// as the original job control

dragJobControl.VerticalAlignment = VerticalAlignment.Top;

dragJobControl.HorizontalAlignment = HorizontalAlignment.Left;

GeneralTransform gt = selectedJobControl.TransformToVisual(JobsGrid);

TranslateTransform trans = new TranslateTransform();

Point p = gt.TransformPoint(new Point(0, 0));

trans.X = p.X;

trans.Y = p.Y;

dragJobControl.RenderTransform = trans;

//add the new jobcontrol on the jobsgrid

JobsGrid.Children.Add(dragJobControl);

//add a recognizer for the new dragging job control

GestureRecognizer recognizer = new GestureRecognizer();

ManipulationInputProcessor manipulationProcessor =

new ManipulationInputProcessor(recognizer, dragJobControl, JobsGrid);

manipulationProcessor.OnPointerPressed(sender, e);

//let the view model know that we are moving this job

jobsViewModel.MoveJob(j);

e.Handled = true;

}

}

1. In the Jobs.xaml file add the event handler to the **DataTemplate** for a Job as shown

<Page.Resources>

<DataTemplate x:Key="JobItemTemplate">

<local:JobControl DataContext="{Binding}" PointerPressed="JobControl\_PointerPressed" />

</DataTemplate>

</Page.Resources>

1. Build and Run the App (**F5**). You can now drag and drop Jobs between columns. You can also try dragging multiple Jobs at the same time. This enables direct manipulation of the content (Jobs) by multiple people working on a big screen (such as a Surface Hub) at the same time.

In this Lab you have learned how to use the **GestureRecognizer** to move, or translate, visual objects around on a Grid. You have improved the UWP app to now accept direct manipulation, this makes it a better app on Surface Hub, and removes the need for the buttons to start and stop a job.