Comparison of Easing and Pointer-Driven Motion Methods – Sixspeedmedia.com 2003, Silverlight Touch processor and Car Game 2001.

The following archival technical comparison documents three distinct pointer-driven motion implementation, along with two related industry-standard examples, preserving the original logic and code integrity.

This comparison focuses exclusively on technical details and actual archived code examples.

Method 1: Car Game (Shockwave/Lingo, January 2001)

Implementation (simplified, archival code):

lingo

CopyEdit

```
on enterFrame
  diffX = (mouseH - sprite.locH)
  moveX = diffX / 10
  sprite.locH = sprite.locH + moveX
end
```

Technical Notes:

- Calculates position adjustments every frame based on pointer location.
- Linear calculation resulting in a naturally smooth, easing-like motion due to progressively smaller movements each frame.

Flick Capability:

- No explicit flick; pointer-follow only.

Method 2: Sixspeedmedia Ellipse and Carousel (Flash/ActionScript, 2003–2005)

Ellipse Carousel Implementation (archival ActionScript):

actionscript

CopyEdit

```
function xmove() {
  homeX = (-_root._xmouse * 1.3) + 300;
  thisX = slideList[0]._x;
  diffX = homeX - thisX;
  moveX = diffX / 50;

  for (var i = 0; i < slideList.length; i++) {
    slideList[i]._x = thisX + moveX + distance * i;
  }
}
setInterval(xmove, 10);
```

Technical Notes:

- Frame-based calculation using an exponential difference division (diffX/factor).
- Results in exponential easing behavior; smoother as the object approaches target.

Flick Capability:

- Implicit natural flick: larger initial diffX results in rapid initial movements ("flick-like") smoothly easing to target.

Method 3: Silverlight Touch Processor (C#/Silverlight, mid-2009, licensed to Microsoft)

Touch & Gesture Implementation (archival licensed C# code):

csharp

CopyEdit

```
private void TranslateElement(Point oldPosition, Point newPosition)
{
    double xDelta = newPosition.X - oldPosition.X;
```

```
    double yDelta = newPosition.Y - oldPosition.Y;

    if (yDelta != 0 || xDelta != 0)
        RaiseTranslate(new TransformEventArgs(TransformType.TRANSLATE, new Point(xDelta,
yDelta)));
}

private void ScaleElement(Point primaryPosition, Point oldPosition, Point newPosition)
{
    double prevLength = GetDistance(primaryPosition, oldPosition);
    double newLength = GetDistance(primaryPosition, newPosition);

    if (prevLength == double.NaN || newLength == double.NaN)
        return;

    double scale = (newLength - prevLength) / newLength;

    if (scale != 0)
        RaiseScale(new TransformEventArgs(TransformType.SCALE, scale));
}
```

Technical Notes:

- Multi-touch capable implementation.
- Position adjustments calculated each frame based on pointer delta.
- Distance-based frame-by-frame recalculation.

Flick Capability:

- Explicit flick capability: delta-based momentum persists briefly after pointer lift, decaying exponentially.

Industry Reference 1: Apple's iPhone Multi-touch (JavaScript/WebKit, circa 2007–2008)

Simplified archival example:

javascript

CopyEdit

```
onTouchMove:
  newLeft = oldLeft + (currentFingerX - lastFingerX);
```

Technical Notes:

- Frame-based direct finger delta calculation.
- Linear pointer-movement without explicit exponential easing logic in developer's JavaScript.

Flick Capability:

- OS-managed momentum; not explicitly coded in JavaScript.

Industry Reference 2: Microsoft WPF Manipulation & Inertia (2009–2010)

Simplified archival logic (pseudo):

pseudo

CopyEdit

```
velocity = initialVelocity - (decelerationRate * elapsedTime)
```

Technical Notes:

- Frame-by-frame linear or polynomial deceleration based explicitly on velocity.
- Explicit velocity-based calculation (rather than distance-based).

Flick Capability:

- Explicit flick/momentum with defined velocity and deceleration parameters.

Direct Technical Comparison (Simplified)

Method	Calculation Type	Easing Type	Pointer Type	Flick?	Multi-touch?
Car Game (Shockwave, 2001)	diffX/frame (linear)	Linear	Mouse	No	No
Sixspeedmedia Carousel (Flash, 2003)	diffX/frame (exponential)	Exponential	Mouse	Implicit	No
Silverlight Processor (C#, 2009)	diff/frame (exponential)	Exponential	Touch/Mouse	Explicit	Yes
Apple iPhone (JavaScript, 2007)	delta/frame (linear)	Linear	Touch	OS-level	Yes
Microsoft WPF (2009–2010)	velocity/frame (linear)	Linear/poly	Touch/Mouse	Explicit	Yes

- **Linear:** constant fraction each frame.
- **Exponential:** shrinking fraction of distance each frame (diffX / constant).
- **Explicit Flick:** intentionally coded momentum.
- **Implicit Flick:** natural motion resembling momentum without explicit velocity calculation.

Cargame Code:

Core Features

1. Realistic Motion Physics

- **Dynamic Speed Adjustment**:
  - Speed (`mygearspeed`) is proportional to user input and gradually adjusted over time using a **friction model**:

    ```
    mygearspeed = mygearspeed * frictionFactor;
    ```

  - Friction ensures smooth deceleration, mimicking real-world motion resistance.
- **Torque and Acceleration**:

- Acceleration and deceleration are influenced by a torque-based formula, providing a responsive and tactile driving experience:

```
torque = (wheelspeed + mygear) / (wheelspeed + 2);
horsepower = (wheelspeed + mygear) * (torque * hp);
```

---

2. Infinite Wrapping Mechanics

- The game implements a **wrapping system** for the yellow road lines and the car itself:
  - Lines that move out of bounds (`<= -200`) are repositioned to the other end (`+700`) for seamless looping:

```
if sprite(me.spritenum).locv <= -200 then
    sprite(me.spritenum).locv = sprite(me.spritenum).locv + 700;
end if;
```

  - This ensures that the stage maintains continuous motion, creating the illusion of an infinite road.

---

3. Visual Feedback and Speed Simulation

- **Yellow Line Compression**:
  - As the car accelerates, the wrapping mechanism causes the yellow road lines to appear closer together, simulating the effect of increased speed.
- **Tachometer and Steering Feedback**:
  - On-screen indicators provide real-time updates on engine speed (`tach`) and steering adjustments:

```
tach = integer(850 + ((speed + 1) / (mygear - tire_grip) /
(torque + 1)) * 600);
```

- **Smooth Steering**:
  - Steering is influenced by speed and gear state, resulting in tighter turns at lower speeds and wider turns at higher speeds:

```
steering = max(-1, min(1, steering));
angle = angle + ((steering * actual_speed) * 0.0005);
```

---

4. Interactivity

- **User Input**:

- Players control speed, steering, and braking using keyboard inputs, which dynamically adjust the car's motion and direction.
- **Dynamic Response**:
  - The car's behavior is affected by real-time calculations of speed, torque, and friction, creating a highly responsive experience.

Significance

1. Early Exploration of Physics-Driven Interactivity

- This project predates many modern physics-based systems, demonstrating a nuanced understanding of motion and tactile feedback in a 2D environment.
- The inclusion of **realistic friction**, **proportional acceleration**, and **wrapping mechanics** set it apart from typical side-scrollers of the time.

2. Foundation for Later Innovations

- The project directly influenced later works, such as the **Sixspeedmedia carousel**, by introducing:
  - **Friction models** for smooth transitions.
  - **Infinite wrapping** for continuous motion.
  - **Proportional motion dynamics** tied to user input.

3. Multi-Platform Compatibility

- Designed for both **in-browser (via Shockwave)** and **standalone (projector)**
- Shockwave enabled rich interactivity in web browsers, while projectors provided high-performance, standalone functionality.

4. Historical Significance

- Developed in January 2001,

Technical Highlights

Feature	Implementation
Friction Model	Dynamic speed decay using `mygearspeed * frictionFactor`.
Infinite Wrapping	Road lines loop seamlessly using boundary checks and repositioning logic.
Dynamic Speed Adjustment	Speed influenced by torque, friction, and user input.

Feature	Implementation
Yellow Line Compression	Faster wrapping creates the visual effect of lines getting closer together.
User Control	Keyboard inputs dynamically adjust speed, steering, and braking.
Responsive Feedback	Real-time updates for tachometer, speed, and steering.

Conclusion

This car game has realistic motion mechanics and interactive feedback. Its wrapping system and tactile motion physics make it a precursor to modern inertia based UI/UX paradigms and interactive carousels.

Global Variables

```
global mygearspeed, speed, wrapHeight, stageHeight
```

- **mygearspeed**: The primary speed affecting the sprite's movement.
- **speed**: A second speed variable (used in `updateUI`), which might be used for display or other logic.
- **wrapHeight**: The amount of vertical space used for sprite wrapping (essentially how far the sprite must travel before it snaps back).
- **stageHeight**: The visible height of the stage (calculated at runtime).

`beginSprite` Handler

```
on beginSprite me
  setupSprite(me)
end
```

- When a Director movie starts (or the sprite first appears on stage), `beginSprite` is called.
- It calls `setupSprite(me)` to initialize sprite properties and set up the global variables.

`setupSprite` Handler

```
on setupSprite me
  global stageHeight, wrapHeight
  spriteRef = sprite(me.spritenum)

  -- Set registration point and calculate stage and wrap dimensions
  spriteRef.member.regPoint = point(0, 0)
  stageHeight = the stageBottom - the stageTop
  wrapHeight = spriteRef.height - stageHeight

  -- Initialize global variables
  mygearspeed = 0
  speed = 0
end
```

1. **Obtain a reference to the current sprite**:

   ```
   spriteRef = sprite(me.spritenum)
   ```

2. **Set the registration point**:

   ```
   spriteRef.member.regPoint = point(0, 0)
   ```

   This makes the top-left corner of the member its reference point.

3. **Calculate stage and wrap dimensions**:

   ```
   stageHeight = the stageBottom - the stageTop
   wrapHeight = spriteRef.height - stageHeight
   ```

   - `stageHeight` is how tall the visible stage area is.
   - `wrapHeight` is how much vertical space above/below is needed so that if the sprite goes out of view, it can "wrap" back around.
4. **Initialize speeds**:

   ```
   mygearspeed = 0
   speed = 0
   ```

   Both speed variables start at zero.

`enterFrame` Handler

```
on enterFrame me
  moveSprite(me)
  applyFriction()
end
```

- **Called every frame** of the Director movie (roughly equivalent to "update" in modern game engines).
- Calls `moveSprite(me)` to shift the sprite's position.
- Calls `applyFriction()` to gradually slow down the speed value.

`moveSprite` Handler

```
on moveSprite me
  spriteRef = sprite(me.spritenum)

  -- Calculate movement amount based on speed
  moveAmount = -mygearspeed / 3
  spriteRef.locv = spriteRef.locv + moveAmount

  -- Wrap sprite if it moves out of bounds
  if spriteRef.locv > 0 then
    spriteRef.locv = spriteRef.locv - wrapHeight
  else if spriteRef.locv < -wrapHeight then
    spriteRef.locv = spriteRef.locv + wrapHeight
  end if
end
```

1. **Get current sprite reference**:

   ```
   spriteRef = sprite(me.spritenum)
   ```

2. **Calculate movement**:

   ```
   moveAmount = -mygearspeed / 3
   spriteRef.locv = spriteRef.locv + moveAmount
   ```

   - The `locv` property is the vertical location of the sprite.

- Dividing `mygearspeed` by 3 makes the sprite move more slowly than the raw speed value might suggest (i.e., a damping effect).
- The negative sign (`-mygearspeed`) inverts the direction if needed.

3. **Vertical wrapping**:

```
if spriteRef.locv > 0 then
  spriteRef.locv = spriteRef.locv - wrapHeight
else if spriteRef.locv < -wrapHeight then
  spriteRef.locv = spriteRef.locv + wrapHeight
end if
```

- If the sprite moves too far down (locv > 0), it repositions the sprite higher up by subtracting `wrapHeight`.
- If it moves too far up (locv < -wrapHeight), it repositions it back down by adding `wrapHeight`.
- This "wraps" the sprite so it continuously scrolls in a loop.

`applyFriction`

```
on applyFriction
  global mygearspeed
  frictionFactor = 0.97  -- Adjust this to control friction
  mygearspeed = mygearspeed * frictionFactor

  -- Ensure speed doesn't drop below a minimum threshold
  if abs(mygearspeed) < 0.1 then
    mygearspeed = 0
  end if
end
```

- **Gradually reduces `mygearspeed`** each frame:

```
mygearspeed = mygearspeed * frictionFactor
```

  This creates a deceleration effect.

- **Minimum threshold**:

```
if abs(mygearspeed) < 0.1 then
  mygearspeed = 0
end if
```

Prevents small floating-point values from never reaching zero, so the sprite can come to a complete stop.

`updateUI`

```
on updateUI
  -- Example: Update text fields or visual indicators
  member(18).text = string(-mygearspeed)
  member(19).text = "Speed: " & string(speed)
end
```

- This is an optional UI update routine.
- Could be called elsewhere in the code (e.g., in `enterFrame` or when needed).
- Shows how you might display the current speed values in text members (e.g., text fields on stage).

Summary

1. **Initialization**: When the sprite starts, `setupSprite` calculates the stage size, sets the sprite's registration point, and initializes the global speed variables.
2. **Movement & Wrapping**: Every frame, `moveSprite` adjusts the sprite's vertical position based on `mygearspeed`, then checks if the sprite is off-screen, wrapping it around if necessary.
3. **Friction**: After movement each frame, `applyFriction` reduces `mygearspeed`, creating a smooth deceleration.
4. **UI Updates**: An optional `updateUI` routine can display or log the current speed values.

In essence, this code continuously moves a sprite in a loop (like a scrolling background). The friction and wrapping give it a smooth, never-ending scrolling effect.

Sixspeedmedia:

Ellipse UI Code

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
////// NOTE: This is COMMENTED FOR CLARITY, ORIGINAL CODE AND LOGIC RETAINED and
archived in original document /////

////////////////////////////////////////////////////////////////////////////////
//////////////////////////////



// disable yellow box around buttons accessed via the keyboard
_focusrect = 0;

// Clip scale at smallest part of ellipse in %
var minScaleInput = 50;
// Clip scale at Largest part of ellipse in %
var maxScaleInput = 250;

//Initalize rotation position in degrees
var rotationinput = 30;

//Set the ellipse -1/2 the height
var ellipseHeightInput = -.5;

// Set width to 100%
var ellipseWidthInput = 1;

// Center reference point Y position
var centerYinput = 250;
// Center reference point X position
var centerXinput = 500;

// Set the size of the ellipse in pixels
var radiusInput = 300;

// Set the initial rotation speed
var speed = .2;

// Number of Clips to create
var numClipsInput = 4;

// Initialize movement speed and start time.
function init() {
    // First, clear the stage of any previous Clips.
    removeClips();
    // Now make a new set of Clips as Int.
    makeClips(parseInt(numClipsInput));
```

```
    // Done initializing, now start the event loop to move the object
    this.attachMovie("processDoMove", "processDoMove", 10);

    //origin is center of logo clips orbit convert to float for precision
    origin._x = parseFloat(centerXInput);
    origin._y = parseFloat(centerYInput);

}

// Function to move a clip with each passing frame.
function moveClipAroundCircle(theClip) {
    // Determine how much time has passed instead of fixed to framerate for
smoother animations
    theClip.then = theClip.now;
    theClip.now = getTimer();
    var elapsed = theClip.now - theClip.then;
    var numSeconds = elapsed / 1000;

   // Calculate the speed of the animation based on the horizontal mouse position
(_xmouse).
// This determines the base value for mouseX, influenced by an external data-
driven speed variable (_root.speed).
mouseX = (_xmouse - 500) * _root.speed;

// Check for extreme conditions in the horizontal mouse position to adjust mouseX
further.
// If the mouse is near the left edge of the screen (_xmouse < 100), slow down
the animation significantly.
if (_xmouse < 100) {
    mouseX = mouseX / 20;
}
// If the mouse is far to the right (_xmouse > 1000), apply a complex adjustment
to reduce the speed dynamically.
// This uses a nested calculation based on _xmouse for diminishing returns as
rudimentary ease out.
// Mimics easing functions by reducing the rate of change as the mouse moves
farther to the right.
else if (_xmouse > 1000) {
    mouseX = mouseX / (_xmouse / (_xmouse / 3));
}

// Override previous adjustments if the mouse is near the far right edge (_xmouse
> 900),
// set idle speed
if (_xmouse > 900) {
```

```
    mouseX = 10;
}

// If the mouse is positioned near the top of the screen (_ymouse < 1),
// set idle speed based on y mouse value
if (_ymouse < 1) {
    mouseX = 10;
}

// Calculate the increment in degrees for this frame based on the adjusted mouseX
value.
// Multiply by the number of seconds elapsed to ensure smooth, time-based
animation.
var degreeIncrement = mouseX * numSeconds;

    // Update the angle
    theClip.theta += degreeIncrement;
    // Keep the angle between -360 and 360 degrees
    theClip.theta %= 360;
    // Convert the ellipse's rotation angle (rotationTheta) from degrees to
radians
    // for use with trigonometric functions (sin, cos) in the following
calculations.
    var radians = degreesToRadians(theClip.theta);
    // Figure out where the clip resides on the ellipse around which it's
travelling.
    // Adjust the ellipse tilt dynamically based on Y mouse position (commented
out).
    //*((_ymouse - 1100) / -700);
    var xOnEllipse = theClip.ellipseWidth * (Math.cos(radians) * theClip.radius);
    var yOnEllipse = theClip.ellipseHeight * (Math.sin(radians) *
theClip.radius);

    // Adjust the calculated position on the ellipse to account for the ellipse's
overall rotation.
    // The raw position (xOnEllipse, yOnEllipse) is rotated by the ellipse's
rotation angle (rotationTheta).
    // This transformation ensures that the ellipse can be tilted or oriented in
any direction, rather than remaining axis-aligned.
    var rotationRadian = degreesToRadians(theClip.rotationTheta);
    var rotatedX = xOnEllipse * Math.cos(rotationRadian) - yOnEllipse *
Math.sin(rotationRadian);
    var rotatedY = xOnEllipse * Math.sin(rotationRadian) + yOnEllipse *
Math.cos(rotationRadian);
```

```
    // Finalize the clip's position by adding the rotated X and Y offsets to the
ellipse's center point.
    // Adjust the Y position with a subtraction because Flash's coordinate system
has an inverted Y-axis.
    theClip._x = theClip.centerX + rotatedX;
    theClip._y = theClip.centerY - rotatedY;

// Simulates a 3d effect

function addDepthEffect(theClip) {
    // Calculate a value using sin(theta), which oscillates between -1 and 1.
    // Adding 1 shifts the range to [0, 2], ensuring all values are positive.
    // These positive values are used for smoothly scaling and fading the clip.
    var degreeSin = Math.sin(degreesToRadians(theClip.theta));
    // Adjust the clip's horizontal and vertical scale.
    // The size oscillates smoothly between minScale and maxScale based on
degreeSin.
    theClip._xscale = theClip.minScale + ((degreeSin + 1) / 2) *
(theClip.maxScale - theClip.minScale);
    theClip._yscale = theClip.minScale + ((degreeSin + 1) / 2) *
(theClip.maxScale - theClip.minScale);
    // Adjust the clip's transparency (alpha).
    // Transparency also oscillates between minScale and maxScale for a fading
effect.
    theClip._alpha = (theClip.minScale + ((degreeSin + 1) / 2) *
(theClip.maxScale - theClip.minScale));
}

// Makes a series of Clips
function makeClips(numClips) {
    // Create an array to store references to all the Clips created.
    ClipList = new Array();

    // Loop through the number of Clips to create and configure each one.
    for (var i = 0; i < numClips; i++) {
        // Attach a new Clip to the stage. The instance name is dynamically
generated as "Clip" + i.
        // The same movie is attached twice, likely an oversight or placeholder
for further logic.
        this.attachMovie("Clip", "Clip" + i, i);
        this.attachMovie("Clip" + i, "Clip" + i, i);

        // Store a reference to the newly attached Clip for easy access.
        var thisClip = this["Clip" + i];
```

```
        // Define properties for the Clip based on user input and default
programmatic values.
        thisClip.radius = parseFloat(radiusInput);         // Radius of the
Clip's motion
        thisClip.centerX = parseFloat(centerXinput);       // Center X position
of the ellipse
        thisClip.centerY = parseFloat(centerYinput);       // Center Y position
of the ellipse
        thisClip.ellipseWidth = parseFloat(ellipseWidthInput);   // Horizontal
scaling of the ellipse
        thisClip.ellipseHeight = parseFloat(ellipseHeightInput); // Vertical
scaling of the ellipse
        thisClip.rotationTheta = parseFloat(rotationInput);      // Initial
rotation angle of the ellipse
        thisClip.maxScale = parseFloat(maxScaleInput);           // Maximum size
of the Clip
        thisClip.minScale = parseFloat(minScaleInput);           // Minimum size
of the Clip

        // Calculate the initial angle (theta) for this Clip, spacing them evenly
around the ellipse.
        thisClip.theta = 0 + (i * (360 / numClips));

        // Record the current time for motion calculations.
        thisClip.now = getTimer();

        // Temporarily position the Clip off-stage to hide it until properly
animated.
        thisClip._x = -1000;
        thisClip._y = -1000;

        // Add the configured Clip to the ClipList array for group management.
        ClipList[i] = thisClip;
    }
}

// Cleans the stage of all Clip clips.
function removeClips() {
    for (var i = 0; i < ClipList.length; i++) {
        ClipList[i].removeMovieClip();
    }
}
// Updates the position and appearance of all Clips in ClipList.
// Each Clip is moved along the ellipse and its size/transparency is adjusted for
a depth effect.
```

```
function moveClips() {
    for (var i = 0; i < ClipList.length; i++) {
        moveClipAroundCircle(ClipList[i]); // Calculate and update the Clip's new
position on the ellipse.
        addDepthEffect(ClipList[i]);      // Adjust the Clip's size and
transparency for a 3D effect.
    }
}
// Converts degrees to radians.
function degreesToRadians(degrees) {
    return (degrees / 180) * Math.PI;
}
stop();
```

2D Carousel Code

```
var startscale = 250;
var endscale = 8;
var distance = 350;
var centerpoint = 500;
var numSlides = 6; // Number of slides
var slideList = []; // Array to hold slide references

// Initialize slideList with slide references
for (var i = 1; i <= numSlides; i++) {
    slideList.push(this["move" + i]);
}

function xmove() {
    homeX = (-_root._xmouse * 1.3) + 300;
    thisX = slideList[0]._x;
    diffX = homeX - thisX;
    moveX = diffX / 50;

    for (var i = 0; i < slideList.length; i++) {
        slideList[i]._x = thisX + moveX + distance * i;
    }

    _root.centerpoint = _xmouse;
    textbox = textval;
}

setInterval(xmove, 10);
```

Silverlight Touch Processor:

Monday, 11 January 2010

Multi-touch in Silverlight 3: Part 1

1. Introduction

1.1 Introduction

One of the cool features in Silverlight 3 is the Multi-touch capabilities. This feature allows Silverlight developers to create fun and interactive applications that can be used with standard inputs, such as mouse and keyboard, as well as with touch-enabled devices.

If you have seen Minority Report, you are already aware of the gesture-based technology. The user can interact with the device to drag objects around, flick to navigate, and pinch to adjust the size of an object. These concepts have become a standard feature on a variety of devices, including Microsoft Surface, Windows 7, and mobile phones.

Silverlight 3 provides the fundamentals of receiving touch events from the hardware to the application. By default, the touch events are fired from Windows 7 to the browser (IE / Firefox). If the application is not processing the touch events, these events are registered as standard mouse events.

Here are two sample applications that utilize Multi-touch concepts. You will need a touch-enabled device and Windows 7 to get the full experience.

3D SlideView | SlideView

In this two-part blog series, I'll show you the process that I went through in developing a Silverlight class library with Multi-touch gestures and events. This process was used in

SharePoint Silverview to add Silverlight / Windows 7 touch functionality in a SharePoint environment. You can find the complete source code at http://code.msdn.microsoft.com/ssv.

1.2 Online Research

Jesse Bishop, Program Manager on the Microsoft Silverlight team, wrote an amazing article about Multi-touch Gesture recognition in Silverlight 3. The article illustrates on how to register touch events and process standard gestures on individual elements. The primary concept is to have a single entity that reads the incoming events and determine which element that is being accessed. After reading Jesse's article, I got inspired to extend his concept with a variety of gestures and events that can be used in a standard Silverlight application.

1.3 Hardware

Before you can develop or run Silverlight application with Multi-touch, you need the right equipment. I have been developing on the Dell SX2210T and the HP TouchSmart IQ586. The HP TouchSmart TX2 is also a good choice. One interesting thing that I learned was the HP TouchSmart IQ586 only had support for a single touch point. There is an updated driver to ensure that the monitor utilizes two touch points. You can check for this by viewing the Pen and Touch information under System Information. It is ideal to have 2 Touch Points, if you plan to support two-finger gestures such as Scale and Rotate. With the recent release of Windows 7, there may be drivers for existing hardware that will need to be updated.

1.4 Touch 101

The Touch framework has two primary concept: the TouchPoint object and the FrameReported event.

The TouchPoint object stores data related to a single touch point. This data is passed from the hardware as the user interacts with the touch device. The number of touch points vary on the supported hardware.   The first touch point that is fired to the application is called the primary touch point. The object has two important properties: Position and Action. The Action property stores the current action state, such as Up, Down, and Move. The Up and Down actions are similar to the mouse's MouseLeftButtonUp and MouseLeftButonDown events. The Move action is the transition action that is constantly called until the user releases the Touch Point, which also invokes the Up action.

The FrameReported event is an application-wide event that sends data from the hardware to the application. One thing to note is that this event is global to the application rather than an individual element. The application doesn't directly know the element that the event is processing. It is possible to retrieve this information using the VisualTreeHelper.GetPointsUnderCoordinates method. The TouchFrameEventArgs argument has methods of getting information about touch points. You can get the current touch points by calling the GetPrimaryTouchPoint and GetTouchPoint methods.

2. Basic Implementation

Demo | Source Code

2.1 Creating the MultiTouch Class Library

Now, let's start by creating a class library. The library will be useful for rapidly developing touch applications. We will only need two objects for the library: TouchElement control and TouchProcessor (explained in Jesse's article).

In Visual Studio 2008 (or Visual Studio 2010 Beta 2), create a new Silverlight Application. Right-click on the Solution in the Solution Explorer and select Add New Project. In the Project Dialog box, select Silverlight Class Library and name it TouchLibrary. Remove the default Class1.cs file.

2.2 TouchElement

We will define the TouchElement control as a content control to allow you to place any element, such as an image, movie clip, or UIElement, into the control. This is not a requirement for deploying Touch applications, but rather a personal choice for development. One useful scenario is having a couple of TouchElement controls with images to allow users to drag and scale images around the user interface.

Right-click on the TouchLibrary and select Add New Item. In the Add New Item Dialog box, select User Control and name it TouchElement. Replace the TouchElement.xaml content with the following:

```
<ContentControl x:Class="TouchLibrary.TouchElement"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  RenderTransformOrigin="0.5,0.5"
  >
  <ContentControl.RenderTransform>
    <TransformGroup>
      <ScaleTransform x:Name="scale" />
      <TranslateTransform x:Name="translate" />
    </TransformGroup>
```

```
  </ContentControl.RenderTransform>
  <ContentControl.Template>
    <ControlTemplate>
      <ContentPresenter Margin="0" Content="{TemplateBinding Content}"
ContentTemplate="{TemplateBinding ContentTemplate}">
      </ContentPresenter>
    </ControlTemplate>
  </ContentControl.Template>
</ContentControl>
```

Note that this is a Content Control rather than a UserControl. We have added TranslateTransform and ScaleTransform transforms to render the control's transformation based on the touch events.

Now it's time for some code action. In Jesse Bishop's article, he implements the concept using ITouchElement as an interface that describes the supported gestures and ManipulateElement as a UserControl that implements the ITouchElement interface. I decided to merge the code from his elements into the TouchElement content control. I highly recommend checking out his article if you want to learn the details of the concept.

We will implement the Translate and Scale gestures. The Translate gesture occurs when the user drags the element from one position to another. The Scale gesture occurs when the user pinch and scales using two touch points. There may be scenarios when the developer wants to create their own custom translate and scale events. For this reason, we will add support for default transforms as well as add custom events that will send information back to the application.

The TransformEventArgs class stores basic data for both Translate and Scale events.

```
public class TransformEventArgs : EventArgs
{
  public TouchElement.TransformType Type { get; set; }
  public Point Translate { get; set; }
  public double Scale { get; set; }

  public TransformEventArgs(TouchElement.TransformType type, Point translate)
  {
    Type = type;
    Translate = translate;
    Scale = 1;
  }

  public TransformEventArgs(TouchElement.TransformType type, double scale)
  {
    Type = type;
    Scale = scale;
    Translate = new Point(0, 0);
  }
}
```

The TransformHandler delegate is used for both events, with the TransformEventArgs object as the second parameter.

```
public delegate void TransformHandler(object sender, TransformEventArgs e);
public event TransformHandler ScaleChanged;
public event TransformHandler TranslateChanged;
```

We will add three properties to the TouchElement class. The TouchEnabled property will be used in the TouchProcessor class to check which TouchElement objects need to be processed. The TouchDrag and TouchScale boolean properties enable the default touch behaviors for drag (translate) and scale events. By default, the properties are set to false to allow the user to create their own events.

```
public bool TouchEnabled
{
  get { return (bool)GetValue(TouchEnabledProperty); }
  set { SetValue(TouchEnabledProperty, value); }
}

public bool TouchDrag
{
  get { return (bool)GetValue(TouchDragProperty); }
  set
  {
    SetValue(TouchDragProperty, value);

    if (value)
      this.TranslateChanged += new TransformHandler(TouchElement_TranslateChanged);
    else
      this.TranslateChanged -= new TransformHandler(TouchElement_TranslateChanged);
  }
}

public bool TouchScale
{
  get { return (bool)GetValue(TouchScaleProperty); }
  set
  {
    SetValue(TouchScaleProperty, value);

    if (value)
```

```
        this.ScaleChanged += new TransformHandler(TouchElement_ScaleChanged);
      else
        this.ScaleChanged -= new TransformHandler(TouchElement_ScaleChanged);
  }
}

public static readonly DependencyProperty TouchEnabledProperty =
  DependencyProperty.Register("TouchEnabled", typeof(bool), typeof(TouchElement), new
PropertyMetadata(true));

public static readonly DependencyProperty TouchDragProperty =
  DependencyProperty.Register("TouchDrag", typeof(bool), typeof(TouchElement), new
PropertyMetadata(false));

public static readonly DependencyProperty TouchScaleProperty =
  DependencyProperty.Register("TouchScale", typeof(bool), typeof(TouchElement), new
PropertyMetadata(false));
```

The TouchDrag and TouchScale properties connect the default event handlers, which validates the input value and directly updates TouchElement’s transform values.

```
void TouchElement_ScaleChanged(object sender, TransformEventArgs e)
{
  double scale = e.Scale;

  double s = this.scale.ScaleX + scale;
  double MIN_SCALE = 0.25;
  double MAX_SCALE = 5.0;

  if (s > MIN_SCALE && s < MAX_SCALE)
  {
```

```
    this.scale.ScaleX += scale;
    this.scale.ScaleY += scale;
  }
  else if (s < MIN_SCALE)
  {
    this.scale.ScaleX = MIN_SCALE;
    this.scale.ScaleY = MIN_SCALE;
  }
  else if (s > MAX_SCALE)
  {
    this.scale.ScaleX = MAX_SCALE;
    this.scale.ScaleY = MAX_SCALE;
  }
}

void TouchElement_TranslateChanged(object sender, TransformEventArgs e)
{
  this.translate.X += e.Translate.X;
  this.translate.Y += e.Translate.Y;
}
```

The TranslateElement and ScaleElement methods calculate the appropriate transforms from the positions based on the touch events. After the value is calculated, the event is fired with the TransformEventArgs as a parameter. This will process the default transformation as well as any custom work that is connected to the event.

```
private void TranslateElement(Point oldPosition, Point newPosition)
{
  double xDelta = newPosition.X - oldPosition.X;
  double yDelta = newPosition.Y - oldPosition.Y;
```

```
    if (yDelta != 0 || xDelta != 0)
        RaiseTranslate(new TransformEventArgs(TransformType.TRANSLATE, new Point(xDelta,
yDelta)));
}

private void ScaleElement(Point primaryPosition, Point oldPosition, Point newPosition)
{
    double prevLength = GetDistance(primaryPosition, oldPosition);
    double newLength = GetDistance(primaryPosition, newPosition);

    if (prevLength == double.NaN || newLength == double.NaN)
        return;

    double scale = (newLength - prevLength) / newLength;

    if (scale != 0)
        RaiseScale(new TransformEventArgs(TransformType.SCALE, scale));
}
```

The TouchProcessor class calls the TouchPointReported method to process the touch event based on the TouchPoint's Action property. On the Down action, we collect information on the TouchPoint. If it is the initial touch point, then the position is later used for distance calculation. In the next blog post, I will go into additional functionality that can be added in the TouchDown method, such as a start time for timing touch event for other gestures. On the Touch Up action, we free up the current point. The TouchUp method can also be used to process any gestures, which will be detailed in the next blog post. The Touch Move is where the fun's at. After the Down action occurs, the Move action constantly runs until the Up action is called. If there is only a single touch point, the TranslateElement method is called. The ScaleElement method is called if there are multiple points. In Jesse's article, he also has Rotate functionality, which is processed with multiple points.

```
public void TouchPointReported(TouchPoint touchPoint)
{
   switch (touchPoint.Action)
   {
      case TouchAction.Down:
         TouchDown(touchPoint.TouchDevice.Id, touchPoint);
         break;
      case TouchAction.Move:
         TouchMove(touchPoint.TouchDevice.Id, touchPoint);
         break;
      case TouchAction.Up:
         TouchUp(touchPoint.TouchDevice.Id, touchPoint);
         break;
   };
}

private void TouchDown(int id, TouchPoint touchPoint)
{
   _points.Add(id, touchPoint.Position);

   if (_points.Count == 1)
   {
      _startPoint = touchPoint.Position;
      _primaryId = id;
   }
}

private void TouchMove(int id, TouchPoint touchPoint)
{
   Point oldPoint = _points[id];
   Point newPoint = touchPoint.Position;

   if ((_points.Count == 1 || id == _primaryId))
      TranslateElement(oldPoint, newPoint);
   else
```

```
      ScaleElement(_points[_primaryId], oldPoint, newPoint);

   _points[id] = newPoint;
}

private void TouchUp(int id, TouchPoint touchPoint)
{
   Point oldPoint = _points[id];
   Point newPoint = touchPoint.Position;
   _points.Remove(id);
}
```

2.3 TouchProcessor

The TouchProcessor class manages the global touch functionality by connecting the Touch's FrameReported event. Jesse goes into detail about the TouchProcessor class in his article. Add the following class to the Class Library. The primary difference in this version is the RootElement property, which is used to connect to the current application's visual root. This property is required for the VisualTreeHelper.FindElementsInHostCoordinates method.

```
public class TouchProcessor
{
   #region Data Members
   private static Dictionary<int,TouchElement> touchControllers = new Dictionary<int,
TouchElement>();
   private static bool _istouchEnabled = false;
   #endregion

   #region Properties
```

```
public static UIElement RootElement
{
  get;
  set;
}

public static bool IsTouchEnabled
{
  get { return _istouchEnabled; }

  set
  {
    if (value && !_istouchEnabled)
      Touch.FrameReported += new TouchFrameEventHandler(Touch_FrameReported);
    else if (!value && _istouchEnabled)
    {
      Touch.FrameReported -= new TouchFrameEventHandler(Touch_FrameReported);
      touchControllers.Clear();
    }

    _istouchEnabled = value;
  }
}
#endregion

#region Constructor
public TouchProcessor()
{ }
#endregion

#region Events
private static void Touch_FrameReported(object sender, TouchFrameEventArgs e)
{
  TouchPointCollection touchPoints = e.GetTouchPoints(null);

  foreach (TouchPoint touchPoint in touchPoints)
  {
    int touchDeviceId = touchPoint.TouchDevice.Id;
```

```
        switch (touchPoint.Action)
        {
          case TouchAction.Down:
            TouchElement hitElement = GetTouchElement(touchPoint.Position);

            if (hitElement != null)
            {
              if (!touchControllers.ContainsKey(touchDeviceId))
                touchControllers.Add(touchDeviceId, hitElement);

              // Prevent touch events from being promoted to mouse events (only if it is over a
TouchElement)
              TouchPoint primaryTouchPoint = e.GetPrimaryTouchPoint(null);
              if (primaryTouchPoint != null && primaryTouchPoint.Action ==
TouchAction.Down)
                e.SuspendMousePromotionUntilTouchUp();

              ProcessTouch(touchPoint);
            }

            break;
          case TouchAction.Move:
            ProcessTouch(touchPoint);
            break;
          case TouchAction.Up:
            ProcessTouch(touchPoint);

            touchControllers.Remove(touchDeviceId);
            break;
        };
      }
    }
    #endregion

    #region Touch Process Events
    private static void ProcessTouch(TouchPoint touchPoint)
    {
```

```
        TouchElement controller;
        touchControllers.TryGetValue(touchPoint.TouchDevice.Id, out controller);

        if (controller != null)
            controller.TouchPointReported(touchPoint);
    }

    private static TouchElement GetTouchElement(Point position)
    {
        foreach (UIElement element in
VisualTreeHelper.FindElementsInHostCoordinates(position, RootElement))
        {
            if (element is TouchElement)
                return (TouchElement)element;

            // immediately-hit element wasn't an TouchElement, so walk up the parent tree
            for (UIElement parent = VisualTreeHelper.GetParent(element) as UIElement; parent !=
null; parent = VisualTreeHelper.GetParent(parent) as UIElement)
            {
                if (parent is TouchElement)
                    return (TouchElement)parent;
            }
        }

        // no TouchElement found
        return null;
    }
    #endregion
}
```

3. Test the TouchLibrary

In the Silverlight Project, add TouchLibrary to the project's References. This will allow you to add the custom touch functionality to your project.

In MainPage.xaml, add the following namespace to connect to the new library.

```
xmlns:touch="clr-namespace:TouchLibrary;assembly=TouchLibrary"
```

Add the following to MainPage.xaml to create three TouchElements (two with color rectangles and one with an image). The image source has the project name pre-pended to ensure that it loads the image from the correct project. The TouchDrag and TouchScale properties are enabled to allow you to manipulate the items.

```
<Grid Background="Black">
  <touch:TouchElement TouchDrag="True" TouchScale="True"
HorizontalAlignment="Left">
    <Rectangle Fill="Red" Width="200" Height="200" />
  </touch:TouchElement>

  <touch:TouchElement TouchDrag="True" TouchScale="True"
HorizontalAlignment="Right">
    <Rectangle Fill="Blue" Width="200" Height="200" />
  </touch:TouchElement>

  <touch:TouchElement TouchDrag="True" TouchScale="True"
HorizontalAlignment="Center" VerticalAlignment="Bottom">
    <Image Source="/TouchApp;component/Rocks.jpg"  Width="200" Height="200" />
```

```
  </touch:TouchElement>
</Grid>
```

In MainPage.xaml.cs, add the following code to initialize the TouchProcessor object.

```
TouchProcessor.RootElement = this;
TouchProcessor.IsTouchEnabled = true;
```

One thing to note is that the Touch functionality doesn't work in Windowless mode. By default, Windowless mode is set to false and it is a feature that should be used with caution. For more information on the limitations of Windowless mode, check out this article.

Compile and run the application. As mentioned before, you will need the required hardware to recognized the Touch events. You can drag the three items across the screen as well as scale them to different sizes. The item is also brought to the front of the screen after release the touch point.

The demo of this version can be found here.

The following is the actual code in the Silverlight processor and gesture library.

```
using System;
using System.Windows;

#nullable disable
namespace MultiTouch
{
  public class TransformEventArgs : EventArgs
  {
    public TouchElement.TransformType Type { get; set; }

    public Point Translate { get; set; }

    public double Scale { get; set; }

    public TransformEventArgs(TouchElement.TransformType type, Point translate)
    {
      this.Type = type;
      this.Translate = translate;
      this.Scale = 1.0;
    }

    public TransformEventArgs(TouchElement.TransformType type, double scale)
    {
      this.Type = type;
      this.Scale = scale;
      this.Translate = new Point(0.0, 0.0);
```

```
    }
  }
}
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

#nullable disable
namespace MultiTouch
{
  public class TouchProcessor
  {
    private static Dictionary<int, List<TouchElement>> touchControllers = new
Dictionary<int, List<TouchElement>>();
    private static bool _istouchEnabled = false;
    private static ChildWindow _childWindow;

    public static UIElement RootElement { get; set; }

    public static bool IsTouchEnabled
    {
      get => TouchProcessor._istouchEnabled;
      set
      {
```

```
        if (value && !TouchProcessor._istouchEnabled)
          Touch.FrameReported += new
TouchFrameEventHandler(TouchProcessor.Touch_FrameReported);
        else if (!value && TouchProcessor._istouchEnabled)
        {
          Touch.FrameReported -= new
TouchFrameEventHandler(TouchProcessor.Touch_FrameReported);
          TouchProcessor._childWindow = (ChildWindow) null;
          TouchProcessor.touchControllers.Clear();
        }
        TouchProcessor._istouchEnabled = value;
      }
    }

    public static ChildWindow PopupWindow
    {
      get => TouchProcessor._childWindow;
      set => TouchProcessor._childWindow = value;
    }

    private static void Touch_FrameReported(object sender, TouchFrameEventArgs e)
    {
      foreach (TouchPoint touchPoint in (PresentationFrameworkCollection<TouchPoint>)
e.GetTouchPoints((UIElement) null))
      {
        int id = touchPoint.TouchDevice.Id;
        switch (touchPoint.Action - 1)
```

```
{
  case 0:
    List<TouchElement> touchElement =
TouchProcessor.GetTouchElement(touchPoint.Position);
    if (touchElement != null)
    {
      if (!TouchProcessor.touchControllers.ContainsKey(id))
        TouchProcessor.touchControllers.Add(id, touchElement);
      TouchPoint primaryTouchPoint = e.GetPrimaryTouchPoint((UIElement) null);
      if (primaryTouchPoint != null && primaryTouchPoint.Action == 1)
        e.SuspendMousePromotionUntilTouchUp();
      TouchProcessor.ProcessTouch(touchPoint);
      break;
    }
    break;
  case 1:
    TouchProcessor.ProcessTouch(touchPoint);
    break;
  case 2:
    TouchProcessor.ProcessTouch(touchPoint);
    TouchProcessor.touchControllers.Remove(id);
    break;
}
}
}
```

```
   private static void ProcessTouch(TouchPoint touchPoint)
   {
     List<TouchElement> touchElementList;
     TouchProcessor.touchControllers.TryGetValue(touchPoint.TouchDevice.Id, out
touchElementList);
     if (touchElementList == null)
       return;
     foreach (TouchElement touchElement in touchElementList)
       touchElement.TouchPointReported(touchPoint);
   }


   private static List<TouchElement> GetTouchElement(Point position)
   {
     bool flag = false;
     List<TouchElement> touchElement = new List<TouchElement>();
     if (TouchProcessor._childWindow == null)
     {
       foreach (UIElement inHostCoordinate in
VisualTreeHelper.FindElementsInHostCoordinates(position,
TouchProcessor.RootElement))
       {
         if (inHostCoordinate is TouchElement && ((TouchElement)
inHostCoordinate).TouchEnabled)
         {
           if (!touchElement.Contains((TouchElement) inHostCoordinate))
             touchElement.Add((TouchElement) inHostCoordinate);
           if (!((TouchElement) inHostCoordinate).IsDependent || touchElement.Count > 1)
```

```
          return touchElement;
        }
        for (UIElement parent = VisualTreeHelper.GetParent((DependencyObject) inHostCoordinate) as UIElement; parent != null; parent = VisualTreeHelper.GetParent((DependencyObject) parent) as UIElement)
        {
         if (parent is TouchElement && ((TouchElement) parent).TouchEnabled)
         {
          if (!touchElement.Contains((TouchElement) parent))
           touchElement.Add((TouchElement) parent);
          if (!((TouchElement) parent).IsDependent || touchElement.Count > 1)
           return touchElement;
          break;
         }
        }
        if (flag)
         break;
       }
      }
      else
      {
       foreach (UIElement inHostCoordinate in VisualTreeHelper.FindElementsInHostCoordinates(position, (UIElement) TouchProcessor._childWindow))
       {
        if (!(inHostCoordinate is Border))
        {
```

```
        int childrenCount = VisualTreeHelper.GetChildrenCount((DependencyObject)
inHostCoordinate);
        if (childrenCount != 0)
        {
          for (int index = 0; index < childrenCount; ++index)
          {
            UIElement child = VisualTreeHelper.GetChild((DependencyObject)
inHostCoordinate, index) as UIElement;
            if (child is TouchElement && ((TouchElement) child).TouchEnabled)
            {
              if (!touchElement.Contains((TouchElement) child))
                touchElement.Add((TouchElement) child);
              if (!((TouchElement) child).IsDependent || touchElement.Count > 1)
                return touchElement;
              break;
            }
          }
        }
      }
    }
  }
  return touchElement.Count == 0 ? (List<TouchElement>) null : touchElement;
 }
}
}
// Decompiled with JetBrains decompiler
\\MultiTouch.dll
```

```
using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Input;

using System.Windows.Media;


#nullable disable

namespace MultiTouch

{

 public class TouchElement : ContentControl, ITouch

 {

  private const int FLICK_ERROR = 50;

  private const int CLICK_TIME = 150;

  private const int FLICK_TIME = 300;

  private Dictionary<int, Point> _points;

  private Point _startPoint;

  private int _primaryId = -1;

  private DateTime _startTime = DateTime.Now;

  private DateTime _lastClickedTime = DateTime.Now;

  private int _clickCount = 0;

  public static readonly DependencyProperty TouchEnabledProperty =
DependencyProperty.Register(nameof (TouchEnabled), typeof (bool), typeof
(TouchElement), new PropertyMetadata((object) true));
```

```
  public static readonly DependencyProperty TouchDragProperty = 
DependencyProperty.Register(nameof (TouchDrag), typeof (bool), typeof (TouchElement), 
new PropertyMetadata((object) false));

  public static readonly DependencyProperty TouchScaleProperty = 
DependencyProperty.Register(nameof (TouchScale), typeof (bool), typeof (TouchElement), 
new PropertyMetadata((object) false));

  public static readonly DependencyProperty FlickModeProperty = 
DependencyProperty.Register(nameof (FlickMode), typeof (TouchElement.FlickDirections), 
typeof (TouchElement), new PropertyMetadata((object) 
TouchElement.FlickDirections.None));

  public static readonly DependencyProperty IsDependentProperty = 
DependencyProperty.Register(nameof (IsDependent), typeof (bool), typeof 
(TouchElement), new PropertyMetadata((object) false));

  internal ScaleTransform scale;

  internal TranslateTransform translate;

  private bool _contentLoaded;


  public bool TouchEnabled

  {

   get => (bool) ((DependencyObject) 
this).GetValue(TouchElement.TouchEnabledProperty);

   set => ((DependencyObject) this).SetValue(TouchElement.TouchEnabledProperty, 
(object) value);

  }


  public bool TouchDrag

  {

   get => (bool) ((DependencyObject) this).GetValue(TouchElement.TouchDragProperty);

   set
```

```
    {
      ((DependencyObject) this).SetValue(TouchElement.TouchDragProperty, (object) value);
      if (value)
        this.TranslateChanged += new
TouchElement.TransformHandler(this.TouchElement_TranslateChanged);
      else
        this.TranslateChanged -= new
TouchElement.TransformHandler(this.TouchElement_TranslateChanged);
    }
  }

  public bool TouchScale
  {
    get => (bool) ((DependencyObject) this).GetValue(TouchElement.TouchScaleProperty);
    set
    {
      ((DependencyObject) this).SetValue(TouchElement.TouchScaleProperty, (object)
value);
      if (value)
        this.ScaleChanged += new
TouchElement.TransformHandler(this.TouchElement_ScaleChanged);
      else
        this.ScaleChanged -= new
TouchElement.TransformHandler(this.TouchElement_ScaleChanged);
    }
  }
```

```
    public TouchElement.FlickDirections FlickMode
    {
      get
      {
        return (TouchElement.FlickDirections) ((DependencyObject)
this).GetValue(TouchElement.FlickModeProperty);
      }
      set => ((DependencyObject) this).SetValue(TouchElement.FlickModeProperty, (object)
value);
    }

    public bool IsDependent
    {
      get => (bool) ((DependencyObject) this).GetValue(TouchElement.IsDependentProperty);
      set => ((DependencyObject) this).SetValue(TouchElement.IsDependentProperty,
(object) value);
    }

    public event TouchElement.ClickHandler Click;

    public event TouchElement.DragHandler Drag;

    public event TouchElement.DragHandler DragCompleted;

    public event TouchElement.FlickHandler Flick;

    public event TouchElement.TransformHandler ScaleChanged;
```

```
  public event TouchElement.TransformHandler TranslateChanged;

  public TouchElement()
  {
    this.InitializeComponent();
    this._points = new Dictionary<int, Point>();
    this._primaryId = -1;
    ((FrameworkElement) this).Loaded += new
RoutedEventHandler(this.TouchElement_Loaded);
  }

  private void TouchElement_Loaded(object sender, RoutedEventArgs e)
  {
  }

  public void TouchPointReported(TouchPoint touchPoint)
  {
    switch (touchPoint.Action - 1)
    {
      case 0:
        this.TouchDown(touchPoint.TouchDevice.Id, touchPoint);
        break;
      case 1:
        this.TouchMove(touchPoint.TouchDevice.Id, touchPoint);
        break;
```

```
   case 2:
    this.TouchUp(touchPoint.TouchDevice.Id, touchPoint);
    break;
 }
}

public void BringToFront()
{
}

private void InterpretSingleTouchGesture(Point oldPosition, Point newPosition)
{
 this.TranslateElement(oldPosition, newPosition);
}

private void InterpretMultiTouchGesture(Point oldPosition, Point newPosition)
{
 this.ScaleElement(this._points[this._primaryId], oldPosition, newPosition);
}

private void TouchDown(int id, TouchPoint touchPoint)
{
 this._points.Add(id, touchPoint.Position);
 if (this._points.Count == 1)
 {
  this._startPoint = touchPoint.Position;
```

```
    this._primaryId = id;
    this._startTime = DateTime.Now;
  }
  this.BringToFront();
}

private void TouchMove(int id, TouchPoint touchPoint)
{
  if (this.IsDependent)
    return;
  Point point = this._points[id];
  Point position = touchPoint.Position;
  if (this._points.Count == 1 || id == this._primaryId)
    this.InterpretSingleTouchGesture(point, position);
  else
    this.InterpretMultiTouchGesture(point, position);
  this._points[id] = position;
}

private void TouchUp(int id, TouchPoint touchPoint)
{
  Point point = this._points[id];
  Point position = touchPoint.Position;
  this._points.Remove(id);
  if (this._points.Count > 0)
    return;
```

```
    int ticks = this.GetTicks();
    if (ticks > 300)
    {
      this.RaiseDragCompleted(new DragEventArgs(position.X - this._startPoint.X, position.Y
- this._startPoint.Y, ticks));
    }
    else
    {
      if (this.ValidateClick(position))
        return;
      this.ValidateFlick(position);
    }
  }

  private int GetTicks() => (int) DateTime.Now.Subtract(this._startTime).TotalMilliseconds;

  private bool ValidateClick(Point newPoint)
  {
    if (this.Click == null)
      return false;
    int ticks = this.GetTicks();
    DateTime now = DateTime.Now;
    double distance = this.GetDistance(newPoint, this._startPoint);
    if (ticks < 150 && distance < 15.0)
    {
      TimeSpan timeSpan = now.Subtract(this._lastClickedTime);
```

```
    if (timeSpan.TotalSeconds > 1.0)
      this._clickCount = 0;
    this._lastClickedTime = now;
    ++this._clickCount;
    if (this._clickCount == 1)
    {
      if (this.Click != null)
      {
        this.RaiseClick(new ClickEventArgs(TouchElement.Clicks.Single));
        return true;
      }
    }
    else if (this._clickCount >= 2)
    {
      this._clickCount = 0;
      if (this.Click != null && timeSpan.TotalMilliseconds < 500.0)
      {
        this.RaiseClick(new ClickEventArgs(TouchElement.Clicks.Double));
        return true;
      }
    }
  }
  return false;
}

private void ValidateFlick(Point newPoint)
```

```
{
  if (this.FlickMode == TouchElement.FlickDirections.None)
    return;
  double num1 = newPoint.X - this._startPoint.X;
  double num2 = newPoint.Y - this._startPoint.Y;
  bool flag1 = Math.Abs(num2) < 50.0 && Math.Abs(num1) > 25.0;
  bool flag2 = Math.Abs(num1) < 50.0 && Math.Abs(num2) > 25.0;
  if (flag1 && flag2)
    return;
  if (this.FlickMode != TouchElement.FlickDirections.Vertical && flag1)
  {
    if (num1 < 0.0)
      this.RaiseFlick(new FlickEventArgs(TouchElement.FlickDirection.West, num1));
    else
      this.RaiseFlick(new FlickEventArgs(TouchElement.FlickDirection.East, num1));
  }
  else if (this.FlickMode != TouchElement.FlickDirections.Horizontal && flag2)
  {
    if (num2 < 0.0)
      this.RaiseFlick(new FlickEventArgs(TouchElement.FlickDirection.North, num2));
    else
      this.RaiseFlick(new FlickEventArgs(TouchElement.FlickDirection.South, num2));
  }
}

private void TouchElement_ScaleChanged(object sender, TransformEventArgs e)
```

```
{
  double scale = e.Scale;
  double num1 = this.scale.ScaleX + scale;
  double num2 = 0.25;
  double num3 = 5.0;
  if (num1 > num2 && num1 < num3)
  {
    this.scale.ScaleX += scale;
    this.scale.ScaleY += scale;
  }
  else if (num1 < num2)
  {
    this.scale.ScaleX = num2;
    this.scale.ScaleY = num2;
  }
  else
  {
    if (num1 <= num3)
      return;
    this.scale.ScaleX = num3;
    this.scale.ScaleY = num3;
  }
}

private void TouchElement_TranslateChanged(object sender, TransformEventArgs e)
{
```

```
    this.translate.X += e.Translate.X;
    this.translate.Y += e.Translate.Y;
  }

  private void TranslateElement(Point oldPosition, Point newPosition)
  {
    double x = newPosition.X - oldPosition.X;
    double y = newPosition.Y - oldPosition.Y;
    if (y == 0.0 && x == 0.0)
      return;
    this.RaiseDrag(new DragEventArgs(x, y, 0));
    this.RaiseTranslate(new TransformEventArgs(TouchElement.TransformType.TRANSLATE,
new Point(x, y)));
  }

  private void ScaleElement(Point primaryPosition, Point oldPosition, Point newPosition)
  {
    double distance1 = this.GetDistance(primaryPosition, oldPosition);
    double distance2 = this.GetDistance(primaryPosition, newPosition);
    if (distance1 == double.NaN || distance2 == double.NaN)
      return;
    double scale = (double) (int) ((distance2 - distance1) / distance2 * 100.0) / 100.0;
    if (scale == 0.0)
      return;
    this.RaiseScale(new TransformEventArgs(TouchElement.TransformType.SCALE, scale));
  }
```

```
    private double GetDistance(Point point1, Point point2)
    {
      try
      {
        return Math.Sqrt((point1.X - point2.X) * (point1.X - point2.X) + (point1.Y - point2.Y) *
(point1.Y - point2.Y));
      }
      catch
      {
        return double.NaN;
      }
    }

    private void RaiseClick(ClickEventArgs e)
    {
      if (this.Click == null)
        return;
      this.Click((object) this, e);
    }

    private void RaiseDrag(DragEventArgs e)
    {
      if (this.Drag == null)
        return;
      this.Drag((object) this, e);
```

```
}

private void RaiseDragCompleted(DragEventArgs e)
{
  if (this.DragCompleted == null)
    return;
  this.DragCompleted((object) this, e);
}

private void RaiseFlick(FlickEventArgs e)
{
  if (this.Flick == null)
    return;
  this.Flick((object) this, e);
}

private void RaiseScale(TransformEventArgs e)
{
  if (this.ScaleChanged == null)
    return;
  this.ScaleChanged((object) this, e);
}

private void RaiseTranslate(TransformEventArgs e)
{
  if (this.TranslateChanged == null)
```

```
    return;
   this.TranslateChanged((object) this, e);
  }

  [DebuggerNonUserCode]
  public void InitializeComponent()
  {
   if (this._contentLoaded)
    return;
   this._contentLoaded = true;
   Application.LoadComponent((object) this, new
Uri("/MultiTouch;component/TouchElement.xaml", UriKind.Relative));
   this.scale = (ScaleTransform) ((FrameworkElement) this).FindName("scale");
   this.translate = (TranslateTransform) ((FrameworkElement) this).FindName("translate");
  }

  public enum TransformType
  {
   TRANSLATE,
   SCALE,
  }

  public enum FlickDirection
  {
   North,
   South,
```

```
    East,
    West,
  }

  public enum FlickDirections
  {
    None,
    Horizontal,
    Vertical,
    Both,
  }

  public enum Clicks
  {
    Single,
    Double,
  }

  public delegate void ClickHandler(object sender, ClickEventArgs e);

  public delegate void DragHandler(object sender, DragEventArgs e);

  public delegate void FlickHandler(object sender, FlickEventArgs e);

  public delegate void TransformHandler(object sender, TransformEventArgs e);
}
```

```
}
// Decompiled with JetBrains decompiler
// Type: MultiTouch.ITouch
// Assembly: MultiTouch, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// MVID: C8A782CC-A3C2-4D91-A862-4120675262E4
// Assembly location:
C:\Users\test\AppData\Local\Temp\Kyjekif\9b564fc12c\MultiTouch.dll

using System.Windows.Input;

#nullable disable
namespace MultiTouch
{
 public interface ITouch
 {
  void TouchPointReported(TouchPoint touchPoint);

  void BringToFront();
 }
}
// Decompiled with JetBrains decompiler
// Type: MultiTouch.FlickEventArgs
// Assembly: MultiTouch, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// MVID: C8A782CC-A3C2-4D91-A862-4120675262E4
// Assembly location:
C:\Users\test\AppData\Local\Temp\Kyjekif\9b564fc12c\MultiTouch.dll
```

```
using System;


#nullable disable
namespace MultiTouch
{
  public class FlickEventArgs : EventArgs
  {
    public TouchElement.FlickDirection Direction { get; set; }


    public int Value { get; set; }


    public FlickEventArgs(TouchElement.FlickDirection dir, double value)
    {
      this.Direction = dir;
      this.Value = (int) value;
    }
  }
}
// Decompiled with JetBrains decompiler
// Type: MultiTouch.DragEventArgs
// Assembly: MultiTouch, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// MVID: C8A782CC-A3C2-4D91-A862-4120675262E4
// Assembly location:
C:\Users\test\AppData\Local\Temp\Kyjekif\9b564fc12c\MultiTouch.dll


using System;
```

```
#nullable disable
namespace MultiTouch
{
  public class DragEventArgs : EventArgs
  {
    public int Ticks { get; set; }

    public int DeltaX { get; set; }

    public int DeltaY { get; set; }

    public DragEventArgs(double x, double y, int ticks)
    {
      this.DeltaX = (int) x;
      this.DeltaY = (int) y;
      this.Ticks = ticks;
    }
  }
}
// Decompiled with JetBrains decompiler
// Type: MultiTouch.ClickEventArgs
// Assembly: MultiTouch, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// MVID: C8A782CC-A3C2-4D91-A862-4120675262E4
// Assembly location: C:\Users\test\AppData\Local\Temp\Kyjekif\9b564fc12c\MultiTouch.dll
```

```
using System;


#nullable disable
namespace MultiTouch
{
  public class ClickEventArgs : EventArgs
  {
    public TouchElement.Clicks ClickMode { get; set; }


    public ClickEventArgs(TouchElement.Clicks mode) => this.ClickMode = mode;
  }
}
```