WPF And User Interactivity Part I: Dealing With Geometries And Shapes

Posted by Bechir Bejaoui in Articles | WPF on April 27, 2010
Tags: Dealing with Geometries and Shapes, wpf, WPF Programming, WPF and User Intractivity

This first article will show the manner to handle elements within the scene. Indeed, WPF provides us a very helpful class, namely the VisualTreeHelper class.

Source: http://www.c-sharpcorner.com/UploadFile/yougerthen/wpf-and-user-interactivity-part-i-dealing-with-geometries-and-shapes/

Reader Level:

Download Files:

- **Graphics01.zip**

To provide a rich WPF graphical application, one must add some advanced interactivities like the mouse related ones. It will be more convenient and very advantageous for an end user to handle the interface easily by just doing some selections, drag and drop actions that will result in business logic not only to copy, cut and paste some text from and to the clipboard but also to handle graphical elements, shapes and geometries within the scene. In those couple of articles, I will demonstrate some techniques helping to leverage those tasks as a part of the WPF technology.

This first article will show the manner to handle elements within the scene. Indeed, WPF provides us a very helpful class, namely the **VisualTreeHelper** class:

http://msdn.microsoft.com/en-us/library/system.windows.media.visualtreehelper_members.aspx

The **VisualTreeHelper** introduces a method that comes with three flavors:

HitTest(Visual, Point)
HitTest(Visual, HitTestFilterCallback, HitTestResultCallback, HitTestParameters)
HitTest(Visual3D, HitTestFilterCallback, HitTestResultCallback, HitTestParameters3D)

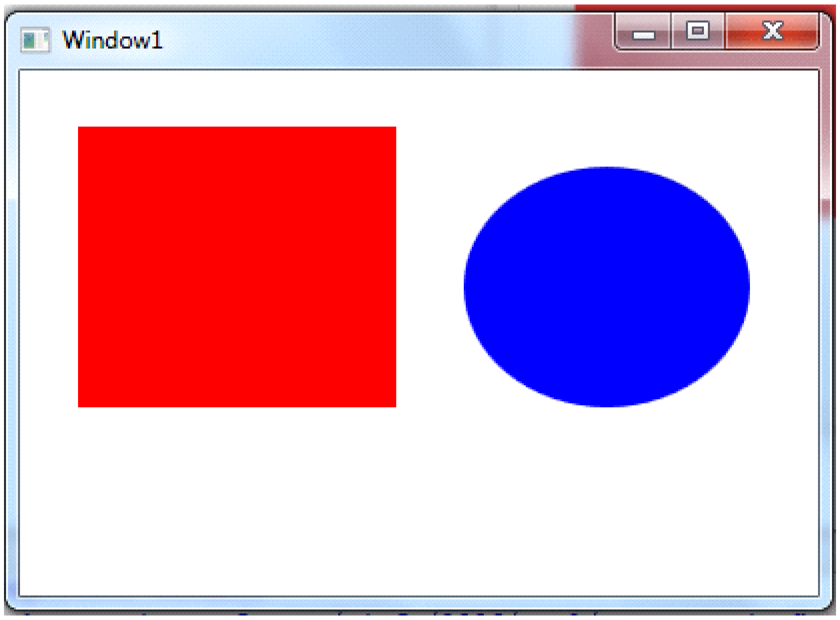
To briefly explain the differences among those three overloads, I will expose those three methods in three different scenarios:

First scenario:

The first one is the simplest one as it requires only two objects as method parameters, namely the hit by the mouse object and the point.

I will propose this XAML:

```
<Canvas Name="canvas1">
<Rectangle Fill="Red" Name="rectangle1"
Height="140" Width="159"
Canvas.Left="29" Canvas.Top="28"/>
<Ellipse Fill="Blue" Name="ellipse1"
Height="120" Width="143"
Canvas.Left="222" Canvas.Top="48" />
</Canvas>
```



We will then implement the code that displays the type of each given object:

```
private void Window_MouseLeftButtonDown(object sender,
MouseButtonEventArgs e)
{

Point hitTest = e.GetPosition(canvas1);
HitTestResult result = VisualTreeHelper.HitTest(canvas1, hitTest);
if (result != null)
{
MessageBox.Show(result.VisualHit.GetType().ToString());
}
}
```

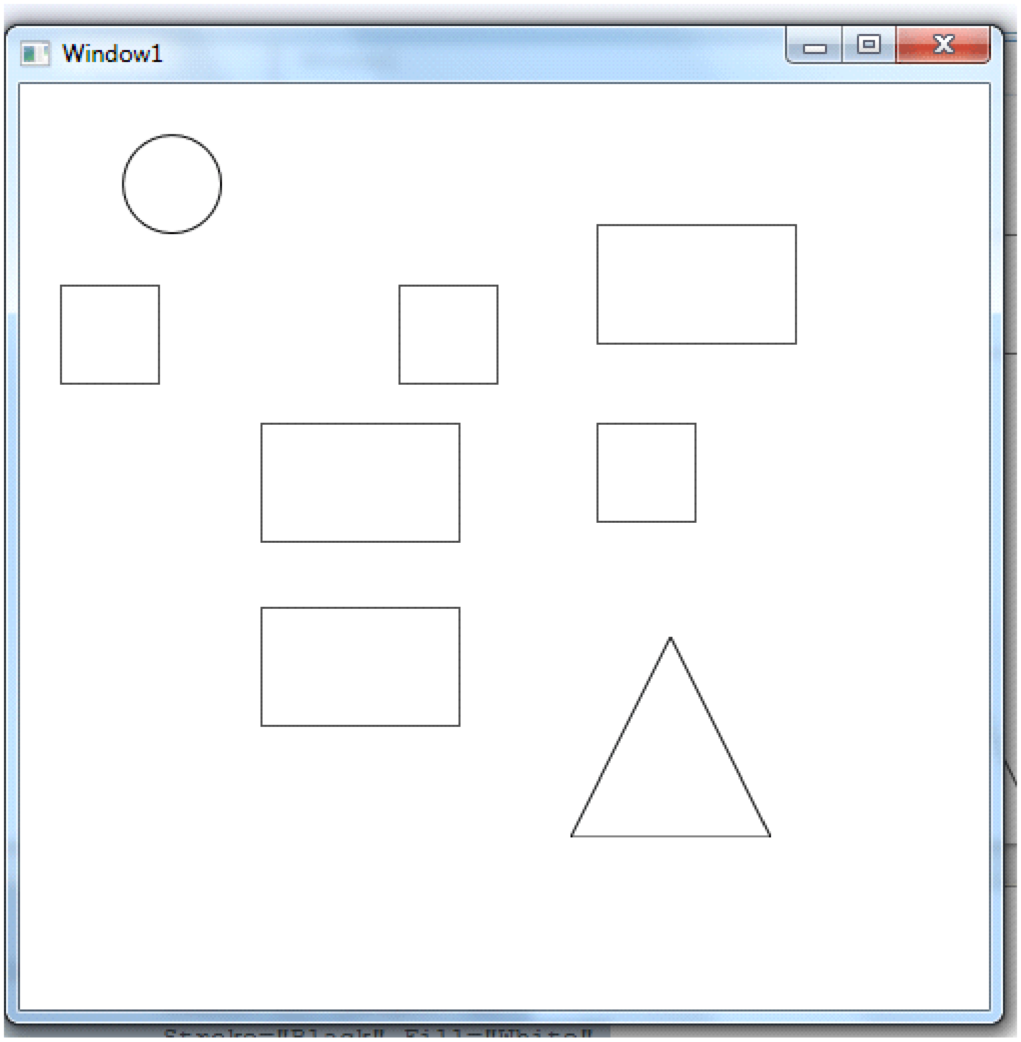
When the user hits the given shape within the mouse then the type of the given object is displayed within a Message box. The resulting object provides a VisualHit; this last one gets the target object as a dependency object. Then you can get the type of that object using GetType method or from the DependencyObjectType property.

The second scenario:

The second one is used for more complex cases, I mean when we have to deal with more than one element on the scene, for that reason, I will propose this XAML code in order to be used in this case:

```
<Canvas x:Name="canvas1"
MouseLeftButtonDown="canvas1_MouseLeftButtonDown">
<Polygon Points="0 100,100 100,50 0"
Stroke="Black" Fill="White" Height="100" Width="100" Canvas.Left="275"
Canvas.Top="276" />
<Ellipse Fill="White" Stroke="Black"
Height="50" Width="50" Canvas.Left="51" Canvas.Top="25" />
<Rectangle Canvas.Left="120" Canvas.Top="261"
Width="100" Height="60"
Stroke="Black" Fill="White"/>
<Rectangle Canvas.Left="120" Canvas.Top="169"
Width="100" Height="60"
Stroke="Black" Fill="White"/>
<Rectangle Canvas.Left="288" Canvas.Top="70"
Width="100" Height="60"
Stroke="Black" Fill="White"/>
<Rectangle Canvas.Left="20" Canvas.Top="100"
Width="50" Height="50"
Stroke="Black" Fill="White"/>
<Rectangle Canvas.Left="288" Canvas.Top="169"
Width="50" Height="50"
Stroke="Black" Fill="White"/>
<Rectangle Canvas.Left="189" Canvas.Top="100"
Width="50" Height="50"
Stroke="Black" Fill="White"/>
</Canvas>
```

This above piece of XAML will help us to get this interface:



Then we have to implement the Mouse left button down event handler of the container which is canvas1 in this case.

But first let's explain the business logic that we will follow to perform the interactivity task.

First, we have to define two objects, one is a generic list that will collect all the elements that the mouse hits within the canvas area. The second is a hit zone that could be simple ellipse geometry in this case.

```
//All shapes will be collected within this list
private List<Shape> hitList = new List<Shape>();
//This will be the hit zone or point
private EllipseGeometry hitArea;
```

The idea is that we capture the mouse position within the target zone which is canvas1 then when the mouse hits a given shape it will be added to the list of elements. In the other hand, the mouse left button down event handler will segregate each kind of shape and apply a particular fill color on it, let's say Blue for the Ellipse, Red for the Rectangle and Yellow for the Polygon which is presented as a triangle in this case.

```
private void canvas1_MouseLeftButtonDown(object sender,
MouseButtonEventArgs e)
{
Initialize();
//You can comment the bellow line to see different behaviour
hitList.Clear();
Point HitPoint = e.GetPosition(canvas1);
hitArea = new EllipseGeometry(HitPoint, 1.0, 1.0);
//This line will call a call back method HitTestCallBack
VisualTreeHelper.HitTest(canvas1, null, HitTestCallBack,
new GeometryHitTestParameters(hitArea));

foreach (Shape item in hitList)
{

if (item is Rectangle)
{
if ((item as Rectangle).Fill == Brushes.White)
{
(item as Rectangle).Fill = Brushes.Red;
}
}
else if (item is Ellipse)
{
if ((item as Ellipse).Fill == Brushes.White)
{
(item as Ellipse).Fill = Brushes.Blue;
}

}
else if (item is Polygon)
{
if ((item as Polygon).Fill == Brushes.White)
{
(item as Polygon).Fill = Brushes.Yellow;
}

}
}
```

The above event handler calls two methods; the Initialize one to initialize all the shapes' colors to white when the mouse left button down is performed.

```
private void Initialize()
{
foreach (Shape shape in canvas1.Children)
{
shape.Fill = Brushes.White;
}
}
```

The second one is the call back method that is called from within the **VisualTreeHelper HitTest** method as a call back method:

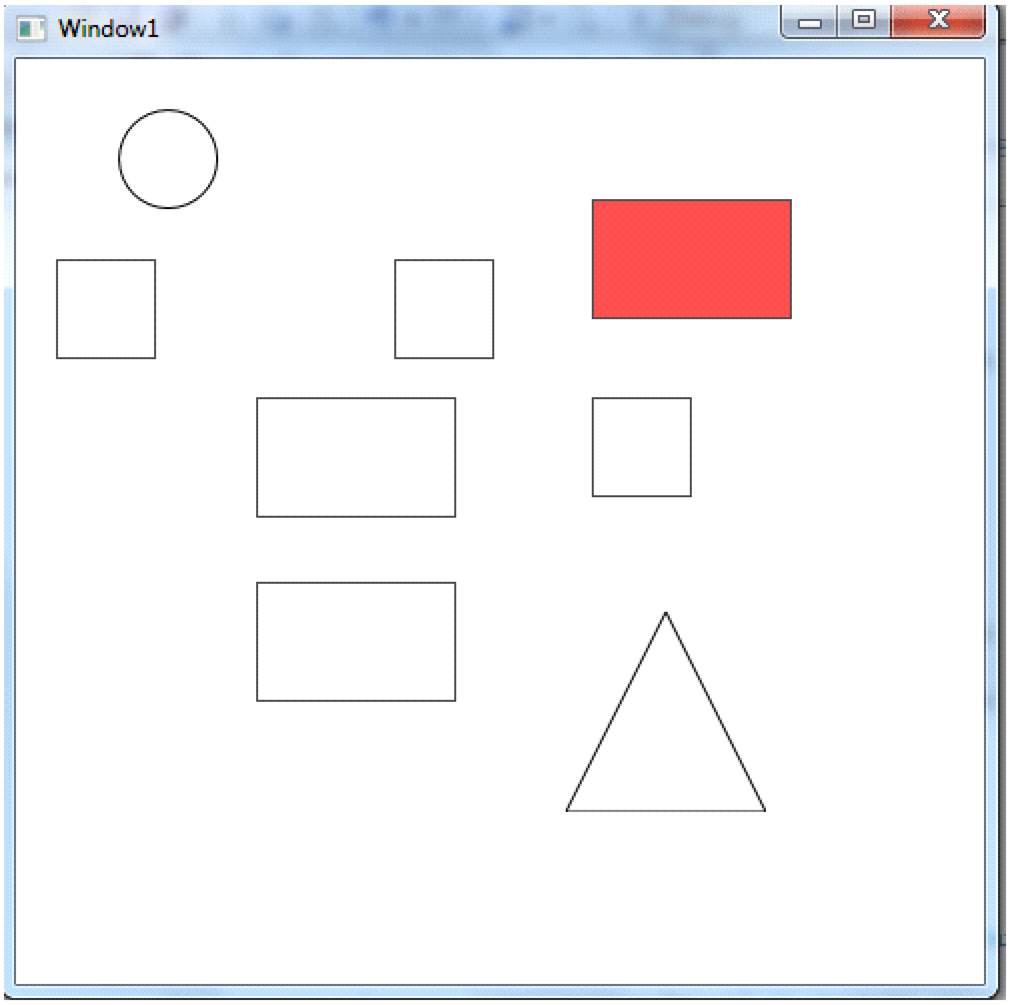
```
private HitTestResultBehavior HitTestCallBack(HitTestResult result)
{
IntersectionDetail intersectionDetail =
(result as GeometryHitTestResult).IntersectionDetail;

switch (intersectionDetail)
{
case IntersectionDetail.FullyContains:
hitList.Add((result.VisualHit as Shape));
return HitTestResultBehavior.Continue;
break;
default:
return HitTestResultBehavior.Stop;
break;
}
}
```

The above call back method accepts a HitTestResult which should be cast to one of two HitTestResults, namely the GeometryHitTestResult or the PointHitTestResult. As we use ellipse geometry as a hit test zone hitArea, we will make use of the first one the GeometryHitTestResult to cast our result. This method will return an object of type HitTestResultBehaviour to tell the VisualTreeHelper HitTest method how the hitArea behaves vis a vis to the Shape position that the mouse hits.

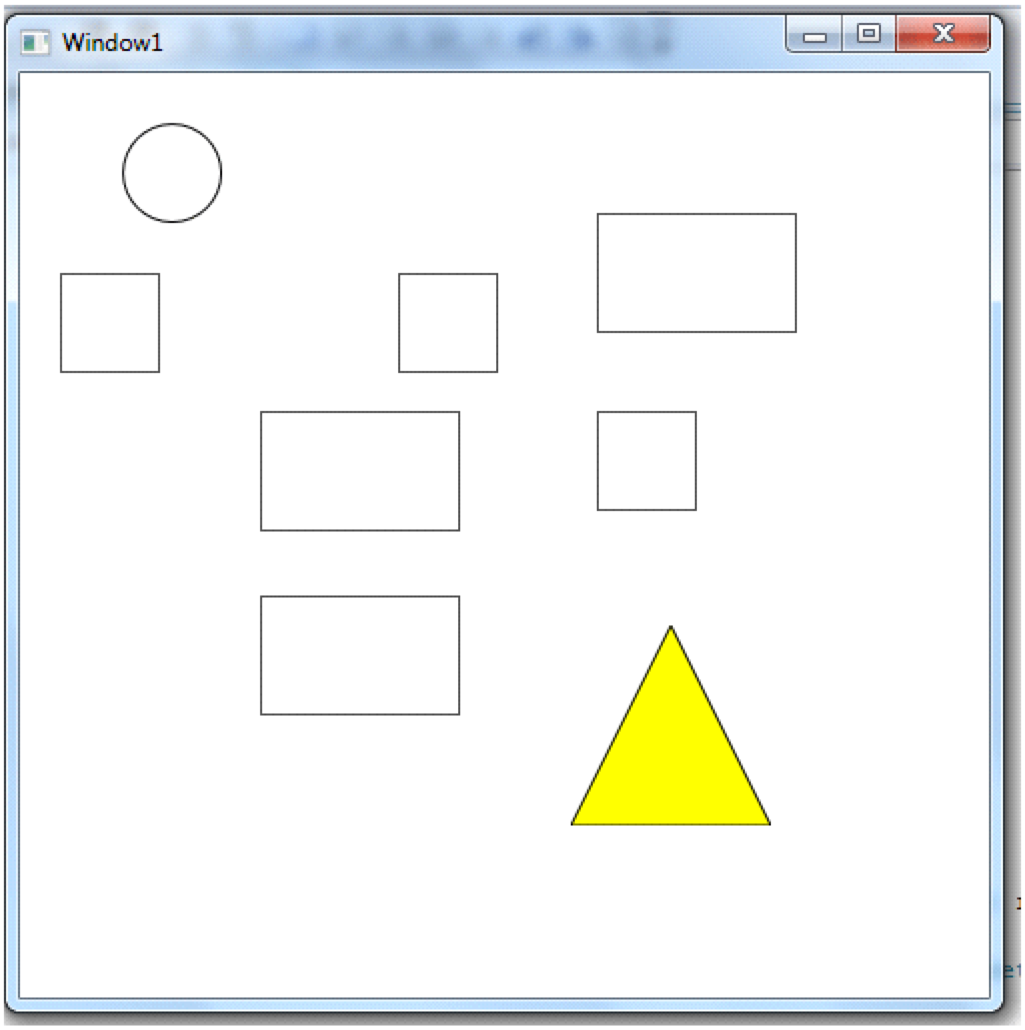
From the code approach, the behavior will be determined according to the IntersectionDetail enumeration which represents the different related positions of the hitArea to the shape that the mouse hits. By the way, it could be represented as follows:

```
IntersectionDetail intersectionDetail = (result as GeometryHitTestResult).IntersectionDetail;
switch (intersectionDetail)
{
case IntersectionDetail.Empty:
/*Write code here to manage the situation
where the hitArea doesn't hit the target
* object at all
*/
break;
case IntersectionDetail.FullyContains:
/*Write code here to manage the situation
where the hitArea is totally contained within
* the target object boundires
*/
break;
case IntersectionDetail.FullyInside:
/*Write code here to manage the situation
where the hitArea totally contains
* the target object
*/
break;
case IntersectionDetail.Intersects:
/*Write code here to manage the situation
where the hitArea intersects with
* the target object
*/
break;
case IntersectionDetail.NotCalculated:
/*Write code here to manage the situation
where the hitArea intersects with
* the target object and this intesction
* is not calculated
*/
break;
default:
break;
}
```

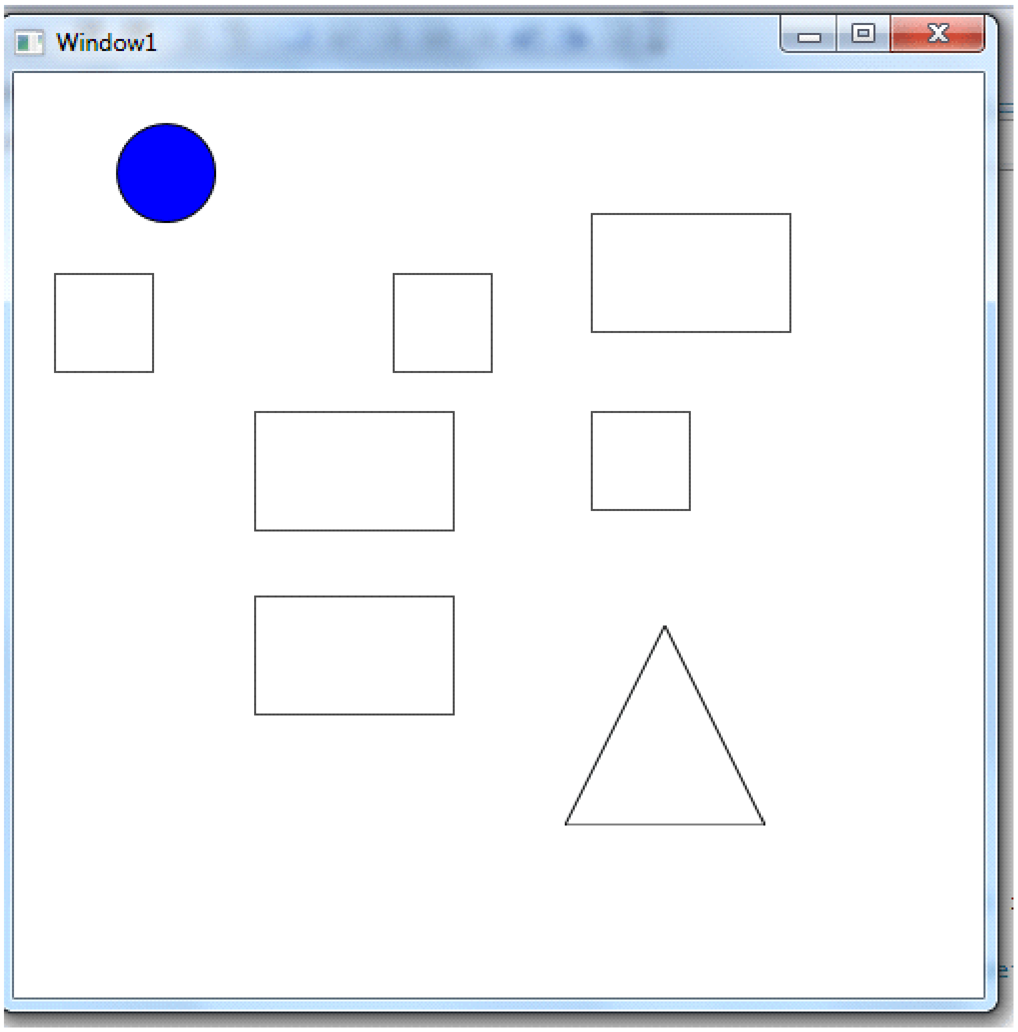
When the rectangle is targeted by the mouse and the user presses the left button, then the rectangle background color will be changed to red.



Then when the triangle is targeted by the mouse, the result will be as below:



And finally when the Ellipse is targeted by the mouse the result will be as below.



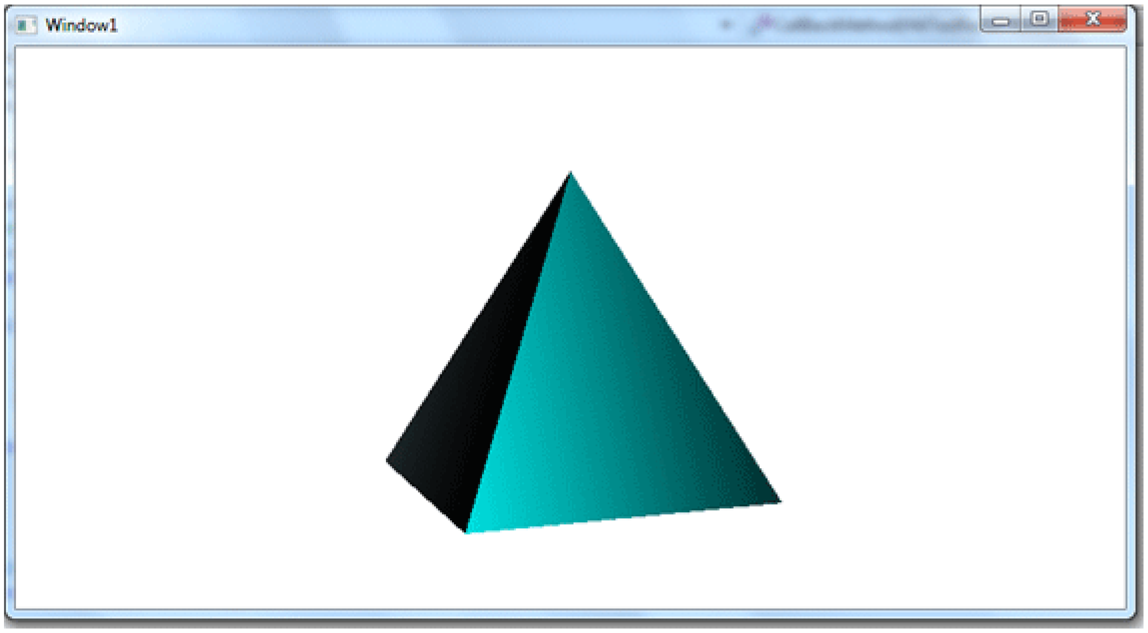
The third Scenario:

It doesn't differ much when comparing with the previous scenario except the fact that our object is of type Visual3D this time, but one important remark should be said in this context is that for this particular overload, the last parameter HitTestParameters3D always causes an InvalidCastException either when using the RayHitTestParameters or RayMeshGeometry3DHitTestResult even though they are types of HitTestParameters3D. But anyway, here is an example of a tree dimensions shape scenario. First let's begin with the shape itself, which will be presented as a pyramid.

```
<Grid Name="LayoutRoot">
<Grid.ContextMenu>
<Viewport3D Name="myViewPort">
<Viewport3D.Camera>
<PerspectiveCamera Position="-3,1,8" LookDirection="3,-1,-8"
UpDirection="0,1,0" FieldOfView="45"
NearPlaneDistance="0.15" />

</Viewport3D.Camera>
<ModelUIElement3D>
```

```
<DirectionalLight Color="White"
Direction="-2,-2,-3" />
</ModelUIElement3D>
<ModelVisual3D>
<ModelVisual3D.Content>
<Model3DGroup x:Name="group3D">

<GeometryModel3D >
<GeometryModel3D.Geometry>
<MeshGeometry3D x:Name="geometry3D"
Positions="0,1,0 1,-1,1 -1,-1,1 1,-1,1 -1,-1,-1"

Normals="0,1,0 -1,0,1 1,0,1 -1,0,-1 1,0,-1"

TriangleIndices="0,2,1 0,3,1 0,3,4 0,2,4" />

</GeometryModel3D.Geometry>

<GeometryModel3D.Material>
<DiffuseMaterial Brush="Aqua" />
</GeometryModel3D.Material>

<GeometryModel3D.BackMaterial>
<DiffuseMaterial Brush="LightBlue"/>
</GeometryModel3D.BackMaterial>
</GeometryModel3D>
</Model3DGroup>
</ModelVisual3D.Content>
</ModelVisual3D>

</Viewport3D>
</Grid>
```

This will provide us this figure when running the code.



Now, the purpose is to simply display the actual coordinate of the hit test point when the user hits the pyramid. To leverage that, we simply make use of the third overload but using the HitTestParameters instead of the HitTestParameters3D which unfortuantly raises an invalid cast exception, for me for all the cases.

```
/// <summary>
/// Raise the mouse left button down event
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Window_Loaded(object sender, RoutedEventArgs e)
{
LayoutRoot.MouseLeftButtonDown +=
new MouseButtonEventHandler(LayoutRoot_MouseLeftButtonDown);
}

/// <summary>
/// This event handler will capture the position on the 3D shape
/// </summary>
/// <param name="sender"></param>
/// <param name="args"></param>
private void LayoutRoot_MouseLeftButtonDown(object sender,
MouseButtonEventArgs args)
{

Point point = args.GetPosition(myViewPort);
VisualTreeHelper.HitTest(myViewPort,null,CallBackMethod,
new PointHitTestParameters(args.GetPosition(myViewPort)));
}
/// <summary>
/// This method will feeds back the hit tests to the HitTest method
/// and it could holds business
/// logic that could be triggered when the hit test is leveraged
/// </summary>
/// <param name="result"></param>
/// <returns></returns>
private HitTestResultBehavior CallBackMethod(HitTestResult result)
{
RayHitTestResult rayResult = result as RayHitTestResult;
if (rayResult != null)
{
RayMeshGeometry3DHitTestResult rayMeshresult =
rayResult as RayMeshGeometry3DHitTestResult;
if (rayMeshresult != null)
{
Point3D point = rayMeshresult.PointHit;
MessageBox.Show(
string.Format("The coordonates of the hit test point
are:\n X:{0}\nY:{1}\nZ:{2}",
point.X.ToString(),
point.Y.ToString(),
point.Z.ToString()));
return HitTestResultBehavior.Stop;
}
}
return HitTestResultBehavior.Continue;
}
```

So let's explain a little bit of the mechanism and what's going on according to this code. First of all, we register the event in the window loaded event:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
LayoutRoot.MouseLeftButtonDown += new
MouseButtonEventHandler(LayoutRoot_MouseLeftButtonDown);
}
```

And then we try to leverage the business logic that it triggered when the user hits the 3D Shape.

```
private void LayoutRoot_MouseLeftButtonDown(object sender,
MouseButtonEventArgs args)
{
Point point = args.GetPosition(myViewPort);
VisualTreeHelper.HitTest(myViewPort,null,
CallBackMethod,
new PointHitTestParameters(args.GetPosition(myViewPort)));
}
```

Of course, the CallBackMethod feeds back the test result in this case to the HitTest method; this first one could also hold some business logic in order to be executed when the user hits the 3D Shape. Indeed, in our case we will add a couple of lines of code within the that method scope to display the test 3D point that the mouse hits, it will be related to the 3D shape boundaries.

The RayHitTestResult is used instead of the HitTestResult to cast the result parameter as we are in 3D context now. In the other hand, the RayMeshGeometry3DHitTestResult will represent the actual 3D Shape for us and then we can get the point hit as the code below shows.

```
private HitTestResultBehavior CallBackMethod(HitTestResult result)
{
RayHitTestResult rayResult = result as RayHitTestResult;
if (rayResult != null)
{
RayMeshGeometry3DHitTestResult rayMeshresult;
rayMeshresult = rayResult as RayMeshGeometry3DHitTestResult;
if (rayMeshresult != null)
{
Point3D point = rayMeshresult.PointHit;
MessageBox.Show(
string.Format("The coordonates of the hit test point
are:\n X:{0}\nY:{1}\nZ:{2}",
point.X.ToString(),
point.Y.ToString(),
point.Z.ToString()));
return HitTestResultBehavior.Stop;
}
}
return HitTestResultBehavior.Continue;
}
```

The first part ends up at this level. I've illustrated a user interactivity use case and particularly using the VisualTreeHelper HitTest result; in the next article, we will remain in the same context but with using more enhanced code techniques.

Good dotneting!!!