Programming with the leap motion

Michaël Van Canneyt

September 22, 2013

Abstract

The leap motion is a small device which acts as a stereographic camera, and which detects movements of hands and fingers. This compact device can be used on small distances, and can be used to implement new modes of interaction with the computer.

1 Introduction

The Leap Motion is a small device (8x3x1 cm) which can be plugged in a USB bus of a computer. It is available from

`http://www.leapmotion.com/`

It acts as an intelligent motion detection sensor: it automatically detects hands, fingers and pointed objects (such as a pen) and tracks the motion of these objects as long as they are in the field of vision of the device. The data is collected by a small service program (delivered with the motion) and is made available to any program using a simple API.

As such, it has a similar operating principle as the Kinect from Microsoft, or the Xtion from Asus. The difference with these latter 2 devices is that it has a finer resolution, and works on small distances from the detector. While the Kinect and Xtion detect complete human bodies or give a stereoscopic view of the surroundings, the Leap Motion focuses on human hands, fingers and pointables. Arguably, the latter are sufficient to control the actions of a computer.

The Leap works on Linux, Windows and Apple MacOS;

The data of the Leap motion can be obtained from a C++ client library, or .NET, Java or Python libraries: the latter bindings are generated using swig from the C++ library. This leaves pascal programmers a bit in the cold, since C++ is not readily usable in a Delphi or Lazarus program.

Luckily, the Leap motion service program also acts as a websocket server, and over the websocket, the same data is presented to any client program as it is presented over the C++ library. This mechanism enables the use of the Leap Motion in a webbrowser - and as of now also in an Object Pascal programming environment.

In this article, a leap motion implementation for Object Pascal is presented. It works in Lazarus/Free Pascal and in Delphi. The bindings have been tested on Linux, Windows. There is no reason to assume they will not work on Apple MacOS out of the box. A `lazleap` package is created for Lazarus, and `delphileapr` and `delphileapd` packages exists for Delphi. These packages will install a websocket leap controller on the component palette.

The sample programs presented here are deliberately kept very simple and graphically not appealing, so the explanation can focus on the working of the Leap and its data structures rather than on complex graphical code.

2 The Leap Motion working principle and Data Model

The leap motion device acts as a camera, and continuously takes stereographic images of the area in front of the sensor. The images are scanned for hands, fingers and pointables (e.g. a pencil). This is done continuously, and as each image is analysed, the results of the analysis are presented to a client program as a `Frame`. A frame is a collection of `Hands` and `Pointable` data: `Fingers` are a special kind of pointable. To be able to track the various hands and pointables in time, as the frames are delivered to the client program, the Leap assigns an numerical ID to each Hand, pointable tool or finger. This ID can be used to match the hands and fingers over the various frames.

The Leap Motion also can detect gestures. It supports detection of 4 gestures:

Swipe A swiping motion, as one would make it on a touch-enabled device.

ScreenTap A horizontal tapping motion, as one would make it on a touch-enabled device.

KeyboardTap A vertictal tapping motion, as one would make it on a keyboard.

Circle If a circle is described using one of the fingers or pointables, this is also detected.

Detection of these gestures can be enabled or disabled: in theory, one could detect these gestures manually, by analysing the evolution of the frames. When gesture detection is enabled, the frames transmitted by the Leap device will also contain gesture data:The `Swipe` and `Circle` gestures are described for some time: they will appear in various data frames as the gesture progresses. The tap gestures are one-shot gestures: their occurence is transmitted only in a single frame.

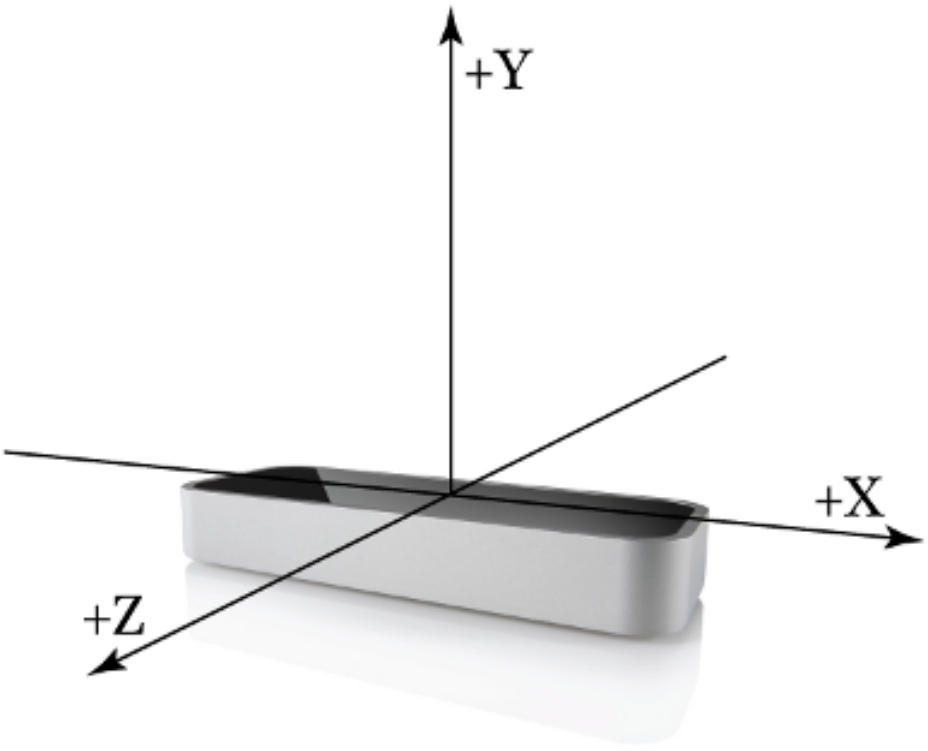
All these objects (hands, fingers, pointables and gestures) are part of a Frame. The leap sends frames as quickly as it can: the number of frames per second (FPS) can vary (the author has recorded speeds between 30 and 60 FPS). Each frame contains several collection of objects: One collection for each type of object. Each object is described by a number of parameters: mostly positions in space. For this, the Leap Motion API uses a 3-dimensional coordinate system, with X,Y and Z axes as shown in figure 1 on page 3: the origin is in the middle of the device, the Y axis is perpendicular to it's surface (vertical, positive values up), and the X axis is along the long axis, while the Z axis is perpendicular to both, positive values increasing away from the screen (if the device is parallel to the screen). The Leap uses the centimeter as the unit of measurement. The coordinates are expressed in Object Pascal using vectors or points:

```
T3DPoint = Record
  X,Y,Z : TFloat;
end;
T3DVector = T3DPoint
```

Each object is desrcibed by a number of vectors or dimensions. First of all, the leap sends with each frame the so-called interaction box: this is a cubic area above the Leap motion sensor, representing the field of view of the sensor. It is described using a center point and 3 dimensions (LxBxH):

```
TInteractionBox = Class
  Property Center : T3DVector;
  Property Dimensions : T3DVector;
  Property Width : TFloat;
  Property Height : TFloat;
  Property Depth : TFloat;
end;
```

Figure 1: The Leap Motion coordinate system



The `Width`, `Height` and `Depth` properties a simple decomposition of the `Dimension` property.

The leap then sends for all pointables (fingers and/or pencils) a `TPointable` data structure:

```
TPointable = Class(TFrameObject)
Public
  Property ID : TLeapID;
  Property Hand : THand;
  Property HandID : TLeapID;
  Property Length : TFloat;
  Property Direction : T3DVector;
  Property TipPosition : T3DPoint;
  Property StabilizedTipPosition : T3DPoint;
  Property TipVelocity : T3DVector;
  Property Width : TFloat;
  Property TouchDist : TFloat;
  Property TouchZone : String;
end;
```

The `ID` is a number assigned to this pointable. This number is unique for each object and remains the same as long as the object remains in the field of vision of the leap. The actual object instances in the API do not remain the same: each new frame contains a collection of new object instances. The biggest part of these properties speak for themselves. The `StabilizedTipPosition` property is a kind of averaged-out position that the leap calculates: the tip can be not always reliably pinpointed, and so the leap attempts to

Figure 2: Finger TipPosition and Direction

interpolate its position.

The `Direction` is the direction in which the pointable it pointing, it is a vector which starts at the tip of the finger (or pencil), as depicted in figure 2 on page 4.

The `TouchDist` and `TouchZone` form an attempt of the Leap Motion to define a touch interaction: The X-Y plane of the Leap forms a 'touch area', and these 2 parameters describe whether the tip of the pointable is close to the 'Touch area': It starts at 1 (pointable enters the touch zone), goes to 0 (touches the touch zone) and then continues to -1 as the pointable penetrates the touch zone. The `TouchZone` string is correlated to this value: one of `none`, `hovering` or `touching` depending on the distance of the pointable to the touch zone.

The `Hand` and `HandID` refer to the hand in which the pointable is held (or, if it concerns a finger: the hand to which the finger belongs). 2 subclasses of `TPointable` exist:

TFinger which describes a finger.

TTool which describes a pointing tool such as a pen.

A hand is described by a number of parameters in the `THand` class:

```
THand = Class (TFrameObject)
  Property Fingers : TFingerList;
  Property Tools : TToolList;
  Property Direction : T3DVector;
  Property PalmNormal : T3DVector;
  Property PalmPosition : T3DVector;
  Property PalmVelocity : T3DVector;
  property SphereCenter : T3DVector;
  property SphereRadius : TFloat;
  Property RotationMatrix : T3DMatrix;
  Property Translation : T3DVector;
  Property Scalefactor : TFloat;
end;
```

The `Fingers` and `Tools` are lists of fingers and tools connected to this hand. The following items describe the hand's position and orientation:

PalmPosition is the position of the center of the hand palm.

PalmNormal is the direction perpendicular to the palm plane.

Direction is the direction from the palm center to the fingers.

SphereCenter describes the curvature of the hand palm and fingers: it is the center of an imaginary ball being held in the hand.

SphereRadius is the radius of the imaginary ball. The smaller the radius, the more curved the hand palm and fingers are.

PalmVelocity is a vector describing the speed of the palm center if the hand is moving.

Translation is a vector describing the translation of the palm center since the last frame.

RotationMatrix is a 3D rotational matrix describing the rotation of the hand since the last frame.

ScaleFactor is a factor desrcribing a change in scale since the last frame (if the hand goes farther from the sensor, it becomes smaller and vice versa).

All this data is collected in a `TFrame` object:

```
TFrame = Class(TIDObject)
  Property ID : TLeapID;
  Property Hands : THandList;
  Property Fingers : TFingerList;
  Property Tools : TToolList;
  Property Gestures : TGestureList;
  Property Pointables : TPointableList;
  Property TimeStamp : Int64;
  Property RotationMatrix : T3DMatrix ;
  Property Translation : T3DVector ;
  Property Scalefactor : TFLoat ;
  Property InteractionBox : TInteractionBox;
end;
```

The `ID` and `TimeStamp` properties serve to identify the frame. The `Hands`, `Fingers` and `Tools` are arrays of detected hands, fingers and tools. The `Gestures` array contains all the detected gestures. `Pointables` is the union of the `Fingers` and `Tools` arrays. Finally, the interactionbox describes, as described earlier, the field of vision of the Leap device.

All objects are defined by an ID. To facilitate searching, the following methods exist:

```
Function Hand(AID : TLeapID) : THand;
Function Finger(AID : TLeapID) : TFinger;
Function Tool(AID : TLeapID) : TTool;
Function Pointable(AID : TLeapID) : TPointable;
Function Gesture(AID : TLeapID) : TGesture;
```

3 The Leap Controller

With the data structures described here, we can already create a program that visualises the data of the leap. But before this can be done, the Leap Controller must be described: this is an actual component that can be installed on the component palette. The Leap Controller component is actually split in 2;

TLeapController the basic abstract leap controller component. Implemented in the `leapdata` unit, together with all the classes that describe the data.

TWebSocketLeapController A leap controller component that fetches data through the websocket API exposed by the Leap Motion.

The `TLeapController` class contains all the needed properties and events to work with the leap controller: it contains the event system, and collects the frames as they are transmitted from the device software.

The `TWebSocketLeapController` is simply a descendent that contains the logic to fetch all data from a websocket: it contains the necessary logic to handle a websocket connection, and to convert the JSON data received from the websocket to the data structures in `leapdata`: TFrame, THand etc.

This split design allows to add a direct client library interface later on without having to change the basic definitions and working code.

```
TLeapController = Class(TComponent)
public
  Property Frames[AIndex :Integer] : TFrame;
  Property FrameCount : Integer;
  Property LastFrame : TFrame;
  Property Version : Integer;
Published
  Property ResolveFrames : Boolean;
  Property EnableGestures : Boolean;
  Property MaxFrames : Integer;
  Property OnFrame : TFrameEvent;
  Property OnVersion : TNotifyEvent;
end;
```

As the frames come in from the Leap device, they are collected by the `TLeapController` component, up to a maximum count of `MaxFrames` frames. When this number is reached, older frames are discarded as newer frames come in, so `FrameCount` is never larger than `MaxFrames`.

Hands and pointables are referred to by `ID`. In particular, a pointable has a `HandID` property and a `Hand` property. The `Hand` property is `Nil` unless `ResolveFrames` is `true`, in which case each new frame will be resolved: the correct `Hand` instances will be looked up and stored as a pointer. This is a relatively time-consuming operation, which can be disabled. The resolving of IDs into actual objects can be done at any moment by calling the `Resolve` method of `TFrame`.

The most important property of this component is the `OnFrame` event. This event is called whenever a new frame is received from the Leap Motion service application. We'll demonstrate it in a small application that draws the tips of the detected fingers on the screen. To do this, we create a form with a paintbox on it. In the form OnCreate event, the websocket version of the leap controller is created and saved in a `FC` variable:

```
  FC:=TWebSocketLeapController.Create(Self);
  FC.OnFrame:=@DoNewFrame;
  FC.OnVersion:=@DoVersion;
  FC.ResolveFrames:=True;
  FC.Enabled:=True;
```

As soon as the `Enabled` property is set to `True` the `TWebSocketLeapController` will create a websocket, and will connect to the websocket. It will connect to the leap

motion service on the default port (6437) and the default host (localhost). This can be changed with the `Hostname` and `Port` properties. The `DoVersion` event is called as soon as the API version is received from the leap socket. The version is displayed in a label:

```
procedure TForm1.DoVersion(Sender: TObject);
begin
  LVersion.Caption:=IntToStr(FC.Version);
end;
```

The more important method is `DoNewFrame`. It receives the new frame as a parameter:

```
procedure TForm1.DoNewFrame(Sender: TObject; AFrame: TFrame);
begin
  Inc(FFrames);
  LFrameCount.Caption:=IntToStr(AFrame.ID);
  Memo1.Lines.BeginUPdate;
  Memo1.Lines.Clear;
  DumpFrame(AFrame,Memo1.Lines);
  Memo1.Lines.EndUpdate;
  FLastFrame:=AFrame;
  PB.Invalidate;
end;
```

It updates the frame count, displays the ID of the frame, and dumps the frame in a memo. After that it saves the frame and invalidates the paintbox `PB`. The paintbox's `OnPaint` handler contains the following code:

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

Var
  P : T3DVector;
  i,R : Integer;
  CX,CY : Integer;
  ICX,ICY,XFactor,YFactor : Double;

begin
  PB.Canvas.Clear;
  if Not Assigned(FLastFrame) then
    Exit;
  PB.Canvas.Brush.Color:=clBlack;
  PB.Canvas.Brush.Style:=bsSolid;
  XFactor:=PB.Width/(FLastFrame.InteractionBox.Width);
  YFactor:=PB.Height/(FLastFrame.InteractionBox.Height);
  ICX:=FLastFrame.InteractionBox.Center.X;
  ICY:=FLastFrame.InteractionBox.Center.Y;
  For I:=0 to FLastFrame.Pointables.Count-1 do
    begin
    P:=FLastFrame.Pointables[i].TipPosition;
    CX:=(PB.Width div 2)+Round((P.X-ICX)*XFactor);
    CY:=(PB.Height div 2)-Round((P.Y-ICY)*YFactor);
    if (P.Z=0) then
      R:=20
    else
```

```
      begin
      R:=Round(600 / Abs(P.Z));
      if R>20 then
        R:=20;
      end;
    PB.Canvas.Ellipse(CX-R,CY-R,CX+R,CY+R);
    end;
end;
```

The routine simply draws a circle for each detected pointable. The circle is drawn at the position the fingertip is at in the X-Y plane, but rescaled so the interactionbox is mapped to the full size of the paintbox. The radius of the circle gets smaller as the tip of the pointable is further removed from the touch zone (the X-Y plane).

4 Gestures

The previous example just uses the finger (or pointable) tip positions to draw something on the screen. The gesture support is not really used. The Leap Motion supports detection of 4 gestures. These are all represented by a common `TGesture` ancestor class:

```
TGesture = Class (TFrameObject)
  Property HandIDS : TLeapIDArray;
  Property PointableIDS : TLeapIDArray;
  Property Hands : THandList;
  Property Pointables : TPointableList;
  Property Duration : Integer;
  Property GestureType : TGestureType;
  Property State : TGestureState;
end;
```

The `HandIDS` and `PointableIDS` are arrays of hand and pointable IDS. They are resolved to `Hands` and `Pointables` when a frame is resolved. Currently, gestures only have 1 hand or pointable, but the data structures are clearly ready for extension to other forms of gestures. The duration is the time (in milliseconds) that the gesture is in progress. The `GestureType` is an enumerated with the following values:

```
TGestureType = (gtUnknown,gtKeyTap,gtScreenTap,gtSwipe,gtCircle);
```

It corresponds to the various supported gesture types. The `GestureState` denotes the state in which the gesture currently is: it is defined as

```
TGestureState = (gsStart,gsUpdate,gsStop);
```

The `gsUpdate` is reported in the case of `gtSwipe` and `gtCircle` gestures. Depending on the type of gesture, a subclass of the `TGesture` class is returned. This is one of the following classes:

TKeyTapGesture for a gesturetype of `gtKeyTap`. This class has additional `Position` and `Direction` properties (of type `T3DPoint` and `T3DVector`), and a `Progress` property (a float). The meaning of these properties is clear from their names.

TScreenTapGesture for a gesturetype of `gtScreenTap`. Just like `TKeyTapGesture`, this class has additional `Position`, `Direction` and `Progress` properties..

TSwipeGesture for a gesturetype of `gtSwipe`. This class also has the `Position` and `Direction` properties, together with a `Speed` and `StartPosition` property.

TCircleGesture for a gesturetype of `gtCircle`. The circle is described by the `Center` and `Radius` properties. The `Progress` property indicates what part of the circle has been completed, with a value of `1` indicating that a complete circle was completed. The `Normal` property describes the normal direction of the plane in which the circle is drawn. The direction of the normal is significant: if the circle is drawn clockwise, then the angle between normal and the pointable will be less than 90 degrees.

5 Tap Gestures

To demonstrate the tap gestures, a small application is made which mimics a point-of-sale application. It shows a series of buttons on the form. Each button is rather large, and displays an image of a product that can be purchased.

The following code creates an array of buttons:

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

Var
  B : TBitBtn;
  I : Integer;
  G : TJpegImage;

begin
  Width:=ILeft+4*(BW+BHS);
  height:=ITop+4*(BH+BVS);
  For I:=0 to 15 do
    begin
    B:=TBitBtn.Create(Self);
    B.Color:=clWhite;
    B.Caption:=FProducts[i];
    G:=TJpegImage.Create;
    try
      G.LoadFromFile(ChangeFileExt(FImages[i],'.jpeg'));
      B.Glyph.Assign(G);
    finally
      G.Free;
    end;
    B.Parent:=Self;
    B.Width:=BW;
    B.Height:=BH;
    B.Tag:=I;
    B.Top:=ITop+(I div 4)*(BH+BVS);
    B.Left:=ILeft+(I mod 4)*(BW+BHS);
    B.Layout:=blGlyphTop;
    B.OnClick:=@DoClick;
    FButtons[i]:=B;
    end;
end;
```

The websocket leap controller is set up in the same way as for the previous program, and

the following Frame event method is made:

```
procedure TForm1.DoFrame(Sender: TObject; AFrame: TFrame);

Var
  P,P2 : TPointable;
  V : T3DVector;
  Pt : TPoint;
  I : Integer;
  B : TBitBtn;
  G : TGesture;


begin
  if AFrame.Pointables.Count=0 then exit;
  P:=AFrame.Pointables[0];
  For I:=1 to AFrame.Pointables.Count-1 do
    begin
    P2:=AFrame.Pointables[i];
    If P.TipPosition.Z>P2.tipPosition.Z then
      P:=P2;
    end;
```

Here, the pointable (finger) with the smallest Z coordinate – or closest to the screen – has been selected: The position of this pointable will be used to select (focus) a button:

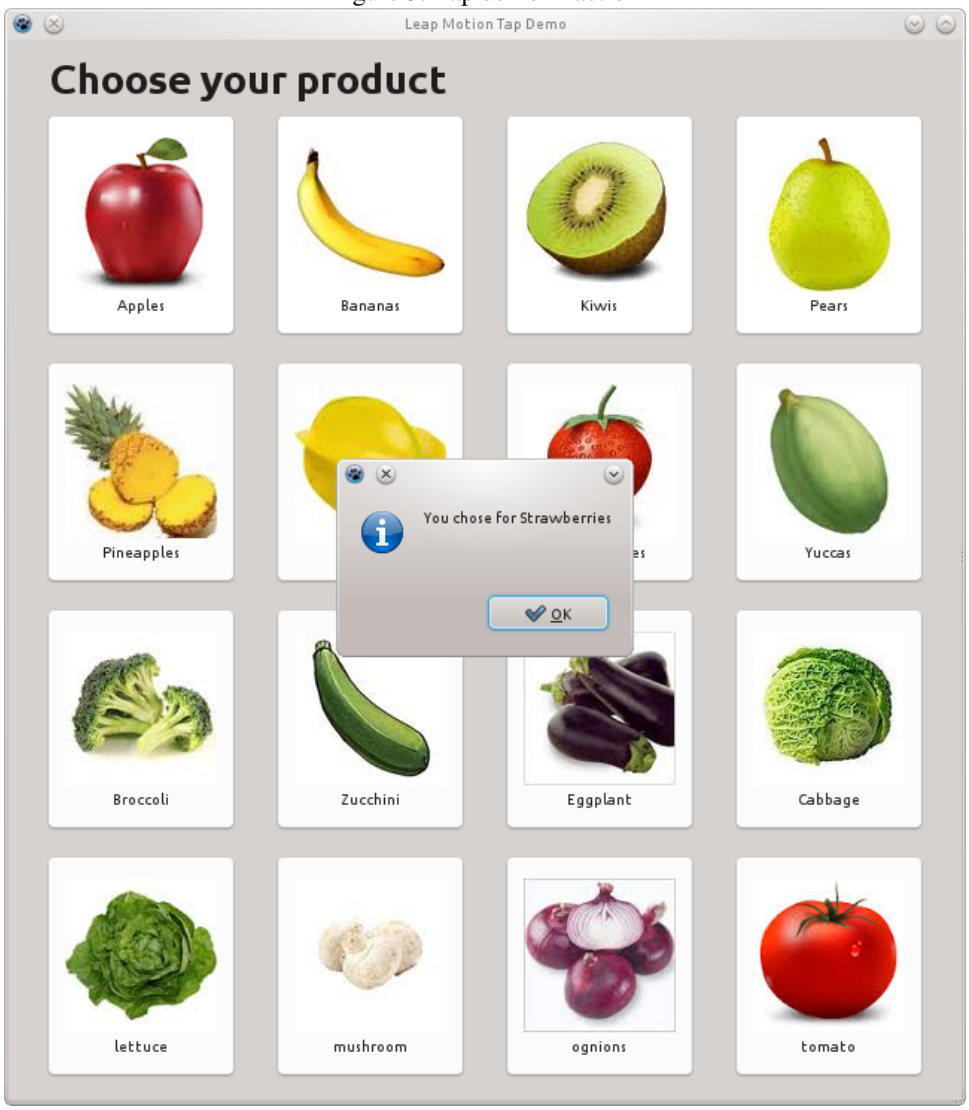
```
  V:=AFrame.InteractionBox.Normalize(P.tipPosition);
  Pt.X:=Round(ILeft+4*V.X*(BW+BHS));
  PT.Y:=Round(ITop+4*(1-V.Y)*(BH+BVS));
  I:=15;
  While (I>=0) and Not (PtInRect(FButtons[i].BoundsRect,Pt)) do
    Dec(I);
  if (I>=0) then
    begin
    FLastButton:=FButtons[i];
    FLastButton.SetFocus;
    end;
```

The coordinates of the tip of the pointable are used to determine a position on the screen, and the button located at that position (if there is one) is focused and saved. The mechanism is rather raw, no rescaling is done as in the previous sample program.

As long as no gesture is reported or no button is focused, this is all that is done: as the finger is moved, the focus changes from button to button. However, as soon as a tap gesture is detected, the focused button is clicked, using the following code:

```
  if (FLastButton=Nil) or (AFrame.Gestures.Count=0) then
    exit;
  for I:=0 to AFrame.Gestures.Count-1 do
    begin
    G:=AFrame.Gestures[i];
    if (AFrame.Gestures[i] is TTapGesture) then
      FlastButton.Click;
    end;
end;
```

Figure 3: Tap demo in action



`TTapGesture` is the common ancestor to `TKeyTapGesture` and `TScreenTapGesture`, so either tap will activate the button.

When the button is clicked, a message is displayed, showing which button was clicked. The result is shown in figure 3 on page 11. The program can use some improvements: the focus is not always visible (this depends on the OS or window manager settings), and the scaling may be improved. The former can be remedied by changing the image on the button, or drawing a nicely visible rectangle around it.

6 Swipe Gestures

The Leap Motion will report a swipe gesture. Ideally, the leap device would be integrated with the operating system GUI (Windows or X-Windows) and swiping would cause grids or listboxes (anything with a scrollbar) to scroll automatically. However, this is not (yet) the case, so this must be mimicked somehow. The following example program attempts to

do so: a small program showing a grid with a list of mp3 files. (simulating a play list of a music player).

The grid is filled from a stringlist containing an author/title/filename trio, separated by a special separator, as follows:

```
Var
  I,J : Integer;
  S : String;

begin
  SGSongs.RowCount:=FList.Count;
  For I:=0 to Flist.Count-1 do
    begin
    S:=Flist[i];
    J:=Pos('#-#',S);
    SGSongs.Cells[0,I]:=Copy(S,1,J-1);
    Delete(S,1,J+2);
    J:=Pos('#-#',S);
    SGSongs.Cells[1,I]:=Copy(S,1,J-1);
    Delete(S,1,J+2);
    SGSongs.Cells[2,I]:=ExtractFileName(S);
    end;
end;
```

There are 2 aspects to a swipe:

1. Detecting the swipe motion itself, and the speed of the swipe. This should start the scrolling of the content.
2. When the swipe is finished, the scrolling should continue, but should gradually slow, and come to a stop.

The start of the swipe is done in the Frame handler of the leap controller:

```
procedure TMainForm.DoFrame(Sender: TObject; AFrame: TFrame);

Var
  I,D : Integer;
  G : TGesture;
  S : TSwipeGesture;

begin
  if (AFrame.Gestures.Count=0) then exit;
  for I:=0 to AFrame.Gestures.Count-1 do
    begin
    G:=AFrame.Gestures[i];
    if (G is TSwipeGesture) then
      begin
      S:=G as TSwipeGesture;
      With S.Direction do
        begin
        If Abs(Y)>Abs(X*2) then
          begin
          D:=Round(Y*S.Speed/10);
```

```
          StartSwipe(StrToIntDef(Trim(MEFriction.Text),1),D,CBPercentual.Checked)
          end;
        end;
      end;
    end;
end;
```

If a swipe gesture is detected, then the swipe direction is examined: if the swipe is roughly vertical a swipe is started. Determining whether a swipe is vertical an be done by examining the X and Y components of the direction: if the Y component is larger than the X component, it can be considered vertical. The factor 5 used in the code is determined through some experiments, but any factory can be taken.

The swipe speed is then used to start a swipe of the grid. The scrolling is controlled through 3 factors:

- An initial speed. This is interpreted as a delta, a number of grid rows to scroll. Negative values will scroll rows up, positive values will scroll rows down.
- A `Friction` parameter, which determines how fast the scrolling speed diminishes.
- A boolean `Percentual`, in which case the friction is interpreted as a percentage. If it is false, the friction is interpreted as an absolute value (causing a lineair slowdown).

The `StartSwipe` routine records the initial values and scrolls the grid:

```
procedure TMainForm.StartSwipe(ASpeed, AFriction : Integer; P : Boolean);

begin
  FFriction:=AFriction;
  FDelta:=ASpeed;
  FPercent:=P;
  DoGridScroll;
  Timer1.Enabled:=True;
end;
```

After the grid was scrolled, a timer is started. The grid scroll starts by scrolling the amount of rows specified in `FDelta`, and then applies the friction:

```
procedure TMainForm.DoGridScroll;

begin
  SGSongs.TopRow:=SGSongs.TopRow+FDelta;
  if not FPercent then
    begin
    If Ffriction>Abs(FDelta) then
      FFriction:=Abs(FDelta);
    If FDelta<0 Then FDelta:=FDelta+FFriction else FDelta:=FDelta-FFriction;
    end
  else
    FDelta:=Trunc(FDelta*(1-(FFriction/100)));
  If FDelta=0 then
    Timer1.Enabled:=False;
end;
```

If the scrolling speed is reduced to zero, the timer is stopped. All the timer does, is scroll the grid:

```
procedure TMainForm.Timer1Timer(Sender: TObject);
begin
  DoGridScroll;
end;
```

The timer interval determines - in combination with the friction parameter, how fast the scrolling stops.

There are quite some parameters that can be used to influence the scrolling: initial speed, the friction algorithm, what to do if a second swipe is detected, etc. Which algorithm produces the nicest effect is something to be investigated.

7 Circle Gestures

When using a finger or pointable to draw a circle in the air, the Leap Motion will detect this, and report a swipe gesture. Naturally, it takes some time for the Leap Motion to detect the circle motion, the circle will be completed already to a certain degree when it is first reported. The `Progress` property indicates what part of the circle has been completed (1 signifies a complete circle). The centre and radius of the circle are reported as well, and the direction of the circle is described by the normal (the perpendicular direction) of the plane in which the circle is drawn.

The circle gesture is demonstrated by creating a small program that rotates an image (turning the image 90 degrees, for simplicity) as a circle gesture is detected. The 4 images are pre-loaded in an array (`FImages`) at program start, to speed up things.

The `DoFrame` handler first attempts to detect a circle gesture. If multiple circle gestures are reported, also attempts to detect the last reported gesture. To this end, it stores the gesture ID (`FLastID`):

```
procedure TForm1.DoFrame(Sender: TObject; AFrame: TFrame);

Var
  I : Integer;
  C,CID : TCircleGesture;
  P : TFLoat;
  IsNew : Boolean;

begin
  if (AFrame.Gestures.Count=0) then exit;
  C:=nil;
  CID:=Nil;
  for I:=0 to AFrame.Gestures.Count-1 do
    if (AFrame.Gestures[i] is TCircleGesture) then
      begin
      C:=AFrame.Gestures[i] as TCircleGesture;
      if (C.ID=FLastID) then
        CID:=C;
      end;
  If (C=Nil) then exit;
```

The second part of this routine checks whether it is a new gesture and whether the gesture is clockwise or not. With these parameters, it calls the `RotateImage` routine:

```
  IsNew:=(CID=Nil);
```

```
  if Not IsNew then
    C:=CID;
  FLastID:=C.ID;

  P:=C.Progress;
  If (C.Pointables.Count>0)
     and Assigned(C.Pointables[0]) then
     begin
     if (AngleTo(C.Pointables[0].Direction,C.Normal)<=Pi/4) then
     // Clockwise, invert progress
      P:=-P;
     end;
  RotateImage(P,IsNew);
end;
```

The `RotateImage` routine does the actual rotating. It is a simple routines, which just calculates the index of the new image to be shown. By varying the factor 4, the sensitivity of the algorithm can be changed. Smaller values will slow down the rotating of the image.

```
procedure TForm1.RotateImage(Progress : TFloat; Reset : Boolean);

Var
  J : integer;

begin
  if Reset then
    FLastProgress:=0;
  J:=Trunc((Progress-FLastProgress)*4);
  If (J<>0) then
    begin
    FLastProgress:=Progress;
    FIndex:=Abs((FIndex+J) mod 4);
    Image1.Picture.Graphic:=FImages[FIndex];
    end;
end;
```

The rotate algorithm is difficult to master, but the idea is to show how the Leap reports circles. Probably it would be better to change the algorithm so that for instance each new circle gesture turns the image 90 degrees. The best solution undoubtedly must be found through some trial and error.

8 Conclusion

The leap is an interesting device. Because of its small size, is can conceivably be built in into keyboards and/or laptop surfaces. As such it could transform the way we interact with a computer, because it focuses on the hands, arguably the part of the human body best suited for the job.

The Leap motion presents us with a very simple yet rich API, demonstrated in several simple programs. The difficulty to unlocking its power lies in finding the right paradigm for interacting with existing or specially designed programs. Looking at traditional touch-enabled devices may give some ideas. Remains that the device is not integrated with the OS's traditional GUI elements. This can be remedied by implementing for example a

component which can be dropped on a form, and which would handle all interaction of the leap with existing GUI controls.