Break-in detection using Lazarus

Michaël Van Canneyt

September 5, 2012

Abstract

A laptop or desktop computer can - using a simple lazarus program - easily be transformed to a burglar alarm. The Windows API allows to create pictures of a room, and using a simple algorithm, changes in the pictures can be detected and an e-mail notification can be sent.

1 Introduction

Most, if not all, laptops or tablets have a webcam. Modern webcams have built-in movement sensors, but also older webcams can easily be used as a motion detector using some simple home-cooked algorithms. This article shows such an algorithm, using Lazarus and Object Pascal.

Lazarus does not have ready-to-use components for accessing webcams. The Lazarus-CCR site does contain a sample program (written by Bogdan Razvan Adrian) that demonstrates using the Windows API for video. For the purpose of this article that sample program will be adapted to serve as a motion detector, which will send an e-mail with an attached image, if motion has been detected.

Windows offers 2 APIs for working with video and webcams. An older, legacy API (Video For Windows) and a new one: DirectShow (part of van DirectX). The new API is much more powerful, but is closer to the hardware and requires the use of callbacks and interfaces. This API is more difficult to use than the older Video For Windows. For simplicity's sake, the older API is used, as it still works under Windows 7.

Sending an e-mail using Lazarus has been discussed in detail in a previous article, and will not be explained again here.

2 Video For Windows

The video for Windows interface is quite simple. A window handle is created, and in this window, the video stream of the webcam is shown. The webcam (or another video source) is controlled by sending Windows messages to the handle of the webcam. There are plenty of messages that can be sent to the webcam window handle, and only a few of them will be used here.

The Video For Windows API has been translated to Object Pascal: the macros that appear in the C/C++ interface, have also been translated: these macros hide the use of messages and present a procedural API. All this is available in the VFW unit, which is available together with the sources that accompany this article. All functions start with the prefix `cap`, from `capture`.

The most important functions (or messages) that are needed for the sample program, are the following:

capCreateCaptureWindow Creates a window handle which will show the camera's video stream. The resulting handle is needed for all other operations described here.

capDriverConnect Connects the window handle with a camera. Up to 10 cameras can be connected.

capDriverDisconnect Disconnects a camera from the window.

capDriverGetCaps Retrieves some of the properties of the camera.

capOverlay Starts or stops the display of the camera image in the window. The display is accomplished through video overlay (i.e. using the hardware).

capPreview Starts or stops the display of the camera image in the window. The display is accomplished through software rendering.

capPreviewRate Sets the framerate of the camera.

capPreviewScale Sets the scale of the camera image.

capGrabFrameNoStop Saves the current frame in a buffer, but does not stop the capturing process.

capFileSaveDIB Saves the frame in the buffer in a .bmp file.

capFileSetCaptureFile Sets a temporary filename for recording a video (.avi).

capCaptureSequence Starts recording a video. The filename of the video must be set using `capFileSetCaptureFile`.

capFileSaveAs Save the recorded video to file.

capCaptureStop Stop recording video.

More functions are available, but the ones described above are sufficient to make a small application. The purpose of the functions is clear, and normally do not need any further explanation. Except for the `capCreateCaptureWindow` function, each function expects the handle of the capture window as the first argument.

3 A sample program

The demonstration application is very simple really: a form with a panel that will display the output of the web camera. Some buttons to start and stop the camera, and to show some default windows dialogs that can be used to set the properties of the camera. There is one button that starts and stops the motion detection algorithm. A status bar to show some status messages completes the user interface.

When the main form of the application is started, the windows video handle is created, using the `capCreateCaptureWindow` function. This function gets the handle of a parent window - in this case the handle of the `pCapture` panel:

```
procedure TMainForm.CapCreate;
begin
  // Destroy if necessary
  CapDestroy;
```

```
  with pCapture do
    FCapHandle := capCreateCaptureWindow('Video Window',
      WS_CHILDWINDOW or WS_VISIBLE or WS_CLIPCHILDREN or WS_CLIPSIBLINGS
      , 5, 5, Width-10, Height-10, Handle, 0);
  if Not CapCreated then
    stCapture.Caption := 'ERROR creating capture window !!!';
end;
```

The capture window handle is created as a child window of the `pcapture` panel using a 5-pixel border. The `CapCreated` function is a method of the `TMainForm` class, it checks whether the `FCapHandle` differs from zero: if the handle is zero, the creation of the video capture window failed.

After the capture window was created, a connection with the webcam driver can be established. This is done in the `CapConnect` method of the form. It uses the `capDriverConnect` function of the VFW API, using the capture window handle as an argument:

```
procedure TMainForm.CapConnect;

Var
  l : integer;
  m : sTRING;

begin
  if Not CapCreated then Exit;
   // Disconnect if necessary
  CapDisconnect;
  // Connect the Capture Driver
  FConnected:=capDriverConnect(FCapHandle, 0);
  if Not FConnected then
    M:='ERROR connecting capture driver.'
  else
    begin
    L:=SizeOf(TCapDriverCaps);
    capDriverGetCaps(FCapHandle,@FDriverCaps,l);
    if FDriverCaps.fHasOverlay then
      M:='Driver connected, accepts overlay'
    else
      M:='Driver connected, software rendering';
    end
  stCapture.Caption:=M;
end;
```

If the driver was connected succesfully, the `capDriverGetCaps` function is used to fetch the driver properties: The `FDriverCaps` record of type `TCapDriverCaps` is filled with the properties of the webcam. This is then used to check whether the camera driver can render the image directly in the video card's memory (`fHasOverlay`): if so, the capability is used, as it works considerably faster.

After the connection with the webcam is made, the actual rendering of the camera's image can start. The rendering is done using the `capPreview` or `capOverlay` functions:

```
procedure TMainForm.CapEnableViewer;

Var
```

```
  M : String;

begin
  FLiveVideo := False;
  if Not FConnected then
    Exit;
  capPreviewScale(FCapHandle, True);     // Allow stretching
  if FDriverCaps.fHasOverlay then        // Driver accepts overlay
    begin
    capPreviewRate(FCapHandle, 0);       // Overlay framerate is auto
    FLiveVideo:=capOverlay(FCapHandle,True);
    M:='Hardware';
    end
  else                                   // Driver doesn't accept overlay
    begin
    capPreviewRate(FCapHandle, 33);      // Preview framerate in ms/frame
    FLiveVideo:=capPreview(FCapHandle, True);
    M:='Software';
    end;
  if FLiveVideo then
    M:=Format('Video Capture - Preview (%s)',[M])
  else
    M:='ERROR configuring capture driver.';
  stCapture.Caption :=M
end;
```

Note that the `capOverlay` or `capPreview` are passed the `True` value. After these functions were called (all this happens in the `OnCreate` event of the main window), the camera is active, and the image is shown in the main form.

The `bReconnect` button also calls these 2 functions - this can be used to activate the camera in case something went wrong.

To stop showing the camera's captured images, the functions `capOverlay` and `capPreview` can again be used. Instead of passing `True`, the value `False` must be passed to stop the display. The `CapDisableViewer` method calls the correct function:

```
procedure TMainForm.CapDisableViewer;
begin
  if FLiveVideo then
    begin
    if FDriverCaps.fHasOverlay then
      capOverlay(FCapHandle,False)
    else
      capPreview(FCapHandle,False);
    FLiveVideo := False;
    end;
end;
```

To record a video, it suffices to call the `capFileSetCaptureFile`, `capCaptureSequence` and `capFileSaveAs` functions. During the recording, it is advisable to stop displaying the captured image for performance reasons. This happens using the above `capDisableViewer` method.

As a filename, a name is created that contains a timestamp:

```
procedure TMainForm.CapRecord;
```

```
Const
  FN = '"Clip-"yyyy-mm-ss-hh-nn-ss".avi"';

begin
  // Stop if not yet stopped.
  CapStop;
  CapDisableViewer;
  // Construct filename
  FFileName:=ExtractFilePath(Application.ExeName);
  FFileName:=FFileName+FormatDateTime(FN,Now);
  stCapture.Caption:='Recording '+FFileName;
  bRecord.Caption := 'S&top';
  // Set filename
  capFileSetCaptureFile(FCapHandle,PChar(FFileName));
  // Start recording
  capCaptureSequence(FCapHandle);
  // Save file.
  capFileSaveAs(FCapHandle, PChar(FFileName));
  FRecording := True;
end;
```

Stopping the video record is done using the `capCaptureStop` function. As soon as the recording is stopped, the filename of the video is changed so it also contains the end time of the recording, and the image of the camera is again showed on screen:

```
procedure TMainForm.CapStop;

Const
  FN = '"---"yyyy-mm-ss-hh-nn-ss".avi"';

Var
  RFN : String:

begin
  if Not FRecording then
    Exit;
  FRecording := False;
  // Stop recording
  capCaptureStop(FCapHandle);
  // Rename file with timestamp
  RFN:=ChangeFileExt(FFileName, FormatDateTime(FN,Now));
  RenameFile(FFileName, RFN);
  // Show preview again on screen
  CapEnableViewer;
  stCapture.Caption := 'Recording stopped';
  bRecord.Caption := '&Record';
end;
```

4 Motion detection

Using all this, the camera can be used to create video, and store it on disk. But how to use the camera as a motion detection device ?

The camera API of Video For Windows can also save the current frame as an image. By doing this at regular intervals, and checking the consecutive images for a meaningful difference, motion can be detected. Once motion is detected, a mail with the image can be sent. To avoid sending too many mails, at most one mail is sent per minute.

To do this, a timer is needed (`TMotion`). The timer is initially disabled, and a push on a button activates the timer. The timer event contains the following code:

```
procedure TMainForm.TMotionTimer(Sender: TObject);

begin
  Inc(FTicks);
  SaveTempFrame;
  if CheckDifferent then
    begin
    If MinutesBetween(Now,FLastSend)>1 then
      begin
      FLastSend:=Now;
      SendPicture;
      end;
    end;
end;
```

`FTicks` is a counter. The `SaveTempFrame` function writes the current camera frame to file. The `CheckDifferent` function checks whether there is a previous image, and returns `True` if there is significant difference between the previous and current image. If a difference (and hence motion) is detected, a mail is sent if at least a minute has elapsed since the last mail.

The interesting functions are `SaveTempFrame` and `CheckDifferent`. The first is quite simple:

```
Procedure TMainForm.SaveTempFrame;

begin
  capGrabFrameNoStop(FCapHandle);
  capFileSaveDIB(FCapHandle,PChar(FFrameFile));
end;
```

`FFrameFile` is the file name, calculated when the program starts.

The `CheckDifferent` function is the hardest part of the program: It must compare the image that was saved in `SaveTempFrame` and compare it with the previous one.

This happens by converting the pixels of the image in grayscale values and compare it pixel by pixel with the previous image, but only if there was one: obviously, the first time there will not be a previous image. The grayscale value is calculated by taking the average of the R,G,B values of the color.

The difference between 2 consecutive images can be expressed in 2 ways: the number of pixels that differ, or the difference in grayscale values can be calculated.

Just counting the number of different pixels gives bad results: The colors in the images that the camera captures, fluctuate: if the grayscale values of 2 consecutive images are compared pixel by pixel, this results almost always in 100% different images. No 2 pixels (in the same location) remain the same. Creating a small statistic shows that the grayscale values fluctuate up to 5% for a stationary image. This fact can be taken into account: When counting differing pixels, 2 pixels are only considered different if they differ more than 5%.

Once the number of different pixels in consecutive images has been counted, a decision needs to be made whether the difference is meaningful. Some experimenting shows that movement in front of the camera results in at least 10% different pixels.

Putting all this together shows that there are 2 parameters for the algorithm:

- The fluctuation allowed between 2 grayscale values of a pixel to consider them different.
- The relative number of differing pixels between 2 images for 2 images to be considered different.

The main form contains 2 spinedits that allow to set these 2 values (measured in %). These percentual values are converted to absolute values in at the start of the `CheckDifferent` function.

The algorithm starts by loading the image in a temporary bitmap, and allocates an array for the grayscale values. Colors in FPC images are a record of word-sized R,G,B values, so the array contains word-sized values for the grayscale values.

```
function TMainForm.CheckDifferent : boolean;

Const
  MaxColor = Cardinal($FFFF);

Var
  A : Array of Word;
  R,C,I,PD,DC,TH,TC : Integer;
  D,MD: Int64;
  G : Word;
  P : TFPColor;

begin
  Result:=Length(FLastImage)<>0;
  FTempBMP.LoadFromFile(FFrameFile);
  TC:=FTempBMP.Height*FTempBMP.Width;
  TH:=Round(MaxColor/100*SETreshold.Value);
  MD:=TC*MaxColor;
  SetLength(A,TC);
```

Here, `MD` is the maximal difference between 2 images ($FFFF multiplied by the number of pixels). `TH` is the minimal difference in color between 2 pixels for them to be considered different. `FLastImage` is the array of grayscale values of the previous image.

After this, the loop for comparing the pixels can be started. For each pixel, a grayscale value is calculated, and saved in the image. At the same time, the difference with the previous grayscale of the pixel is calculated and added to the total difference. If the pixel is considered different, the total amount of different pixels is also increased.

```
  I:=0;
  D:=0;
  dc:=0;
  For R:=0 to FTempBMP.Height-1 do
    For C:=0 to FTempBMP.Width-1 do
      begin
      P:=FTempBMP.Colors[C,R];
```

```
      G:=(P.blue+P.red+P.Green) div 3;
      P.Blue:=G;
      P.Red:=G;
      P.Green:=G;
      FTempBMP.Colors[C,R]:=P;
      A[i]:=G;
      if (I<Length(FLastImage)) then
        begin
        PD:=Abs(G-FLastImage[i]);
        If (PD>TH) then
          begin
          inc(DC);
          D:=D+Abs(PD);
          end;
        end;
      Inc(i);
      end;
```

When the loop is done, the array of grayscale values is saved in `FLastImage`, and the result of the function is calculated. Some statistics are shown in the status bar: the number of different colors, and the number of different pixels. If the function result indicates that the image is different, the image (now transformed to a grayscale image) is saved to disk:

```
  FLastImage:=A;
  STCapture.Caption:=Format('Try %d - Color:  %d (%f %%) Pixels: %d/%d (%f %%)',
                            [FTicks, D, D/MD*100, DC, TC, DC/TC*100]);
  if Result then
    begin
    Result:=(D/MD*100)>SETrigger.Value;
    if Result then
      FTempBMP.SaveToFile(FBWFrameFile);
    end;
end;
```

All that needs to be done now is send the saved image to a mail address. This is done using synapse. The workings of synapse have been explained in a previous article, so the function `SendPicture` can be easily understood:

```
procedure TMainForm.SendPicture;

Var
  Mime : TMimeMess;
  P : TMimePart;
  B : Boolean;
  AText,AServer,ATO : String;
  L : TStringList;

begin
  STCapture.Caption:='Sending picture';
  ATO:='editor@blaisepascal.eu';
  AServer:='mail.blaisepascal.eu';
  AText:=FormatDateTime('dd/mm/yyyy hh:nn:ss',Now);
  AText:=Format('Camera detected movement at %s',[AText]);
  Mime:=TMimeMess.Create;
```

```
  try
    Mime.Header.ToList.Text:=ATo;
    Mime.Header.Subject:='Motion detected';
    Mime.Header.From:=ATo;
    P:=Mime.AddPartMultipart('mixed',Nil);
    L:=TstringList.Create;
    try
      L.Text:=AText;
      Mime.AddPartText(L,P);
      Mime.AddPartBinaryFromFile(FFrameFile,P);
      Mime.EncodeMessage;
      B:=SendToRaw(ATo,ATo,AServer,Mime.Lines,'','');
    finally
      L.Free;
    end;
    if not B then
      STCapture.Caption:='Failed to send picture'
    else
      STCapture.Caption:='Sent picture to '+ATo;
  finally
    Mime.Free;
  end;
end;
```

5 conclusion

It is quite simple to record a video using a webcam and Lazarus. Using the webcam as a motion detector is also not hard, as shown in the code of this article. The algorithm presented here is probably not the best algorithm, but it is conceptually simple and understandable. It can be easily adapted: there is room for variation: the grayscale values can be calculated differently, the difference between 2 pixels can also be calculated differently. It is possible to consider only part of the image for comparison. There are probably also better algorithms than a straightforward comparison: Entering the terms `Motion`, `Detection` and `Algorithm` in Google results in a lot of scientific publications on the subject.