Motion Detection Algorithms

Introduction

There are many approaches for motion detection in a continuous video stream. All of them are based on comparing of the current video frame with one from the previous frames or with something that we'll call background. In this article, I'll try to describe some of the most common approaches.

In description of these algorithms I'll use the AForge.NET framework, which is described in some other articles on Code Project: [1], [2]. So, if you are common with it, it will only help.

The demo application supports the following types of video sources:

- AVI files (using Video for Windows, interop library is included);
- updating JPEG from internet cameras;
- MJPEG (motion JPEG) streams from different internet cameras;
- local capture device (USB cameras or other capture devices, DirectShow interop library is included).

Algorithms

One of the most common approaches is to compare the current frame with the previous one. It's useful in video compression when you need to estimate changes and to write only
the changes, not the whole frame. But it is not the best one for motion **detection** applications. So, let me describe the idea more closely.

Assume that we have an original 24 bpp RGB image called current frame (image), a grayscale copy of it (currentFrame) and previous video frame also gray scaled (backgroundFrame). First of all, let's find the regions where these two frames are differing a bit. For the purpose we can use Difference and Threshold filters.

```java
// create filters
Difference differenceFilter = new Difference();
IFilter thresholdFilter = new Threshold(15);
// set background frame as an overlay for difference filter
differenceFilter.OverlayImage = backgroundFrame;
// apply the filters
Bitmap tmp1 = differenceFilter.Apply(currentFrame);
Bitmap tmp2 = thresholdFilter.Apply(tmp1);
```

On this step we'll get an image with white pixels on the place where the current frame is different from the previous frame on the specified threshold value. It's already possible to count the pixels, and if the amount of it will be greater than a predefined alarm level we can signal about a motion event.

But most cameras produce a noisy image, so we'll get motion in such places, where there is no motion at all. To remove random noisy pixels, we can use an Erosion filter, for example. So, we'll get now mostly only the regions where the actual motion was.

```java
// create filter
IFilter erosionFilter = new Erosion();
// apply the filter
Bitmap tmp3 = erosionFilter.Apply(tmp2);
```

The simplest motion detector is ready! We can highlight the motion regions if needed.

```java
// extract red channel from the original image
IFilter extrachChannel = new ExtractChannel(RGB.R);
Bitmap redChannel = extrachChannel.Apply(image);
// merge red channel with motion regions
Merge mergeFilter = new Merge();
mergeFilter.OverlayImage = tmp3;
Bitmap tmp4 = mergeFilter.Apply(redChannel);
// replace red channel in the original image
ReplaceChannel replaceChannel = new ReplaceChannel(RGB.R);
replaceChannel.ChannelImage = tmp4;
Bitmap tmp5 = replaceChannel.Apply(image);
```

Here is the result of it:
From the above picture we can see the disadvantages of the approach. If the object is moving smoothly we'll receive small changes from frame to frame. So, it's impossible to get the whole moving object. Things become worse, when the object is moving so slowly, when the algorithms will not give any result at all.

There is another approach. It's possible to compare the current frame not with the previous one but with the first frame in the video sequence. So, if there were no objects in the initial frame, comparison of the current frame with the first one will give us the whole moving object independently of its motion speed. But, the approach has a big disadvantage - what will happen, if there was, for example, a car on the first frame, but then it is gone? Yes, we'll always have motion detected on the place, where the car was. Of course, we can renew the initial frame sometimes, but still it will not give us good results in the cases where we can not guarantee that the first frame will contain only static background. But, there can be an inverse situation. If I'll put a picture on the wall in the room? I'll get motion detected until the initial frame will be renewed.

The most efficient algorithms are based on building the so called background of the scene and comparing each current frame with the background. There are many approaches to build the scene, but most of them are too complex. I'll describe here my approach for building the background. It's rather simple and can be realized very quickly.

As in the previous case, let's assume that we have an original 24 bpp RGB image called current frame (image), a grayscale copy of it (currentFrame) and a background frame also gray scaled (backgroundFrame). At the beginning, we get the first frame of the video sequence as the background frame. And then we'll always compare the current frame with the background one. But it will give us the result I've described above, which we obviously don't want very much. Our approach is to "move" the background frame to the current frame on the specified amount (I've used 1 level per frame). We move the background frame slightly in the direction of the current frame - we are changing colors of pixels in the background frame by one level per frame.
MoveTowards moveTowardsFilter = new MoveTowards();
// move background towards current frame
moveTowardsFilter.OverlayImage = currentFrame;
Bitmap tmp = moveTowardsFilter.Apply(backgroundFrame);
// dispose old background
backgroundFrame.Dispose();
backgroundFrame = tmp;

And now, we can use the same approach we've used above. But, let me extend it slightly to get a more interesting result.

// create processing filters sequence
FiltersSequence processingFilter = new FiltersSequence();
processingFilter.Add(new Difference(backgroundFrame));
processingFilter.Add(new Threshold(15));
processingFilter.Add(new Opening());
processingFilter.Add(new Edges());
// apply the filter
Bitmap tmp1 = processingFilter.Apply(currentFrame);

// extract red channel from the original image
IFilter extractChannel = new ExtractChannel(RGB.R);
Bitmap redChannel = extractChannel.Apply(image);

// merge red channel with moving object borders
Merge mergeFilter = new Merge();
mergeFilter.OverlayImage = tmp1;
Bitmap tmp2 = mergeFilter.Apply(redChannel);

// replace red channel in the original image
ReplaceChannel replaceChannel = new ReplaceChannel(RGB.R);
replaceChannel.ChannelImage = tmp2;
Bitmap tmp3 = replaceChannel.Apply(image);

Now it looks much better!

There is another approach based on the idea. As in the previous cases, we have an original frame and a gray scaled version of it and of the background frame. But let's
apply `Pixellate` filter to the current frame and to the background before further processing.

```csharp
// create filter
IFilter pixellateFilter = new Pixellate();
// apply the filter
Bitmap newImage = pixellateFilter(image);
```

So, we have pixellated versions of the current and background frames. Now, we need to move the background frame towards the current frame as we were doing before. The next change is only the main `processing` step:

```csharp
// create processing filters sequence
FiltersSequence processingFilter = new FiltersSequence();
processingFilter.Add(new Difference(backgroundFrame));
processingFilter.Add(new Threshold(15));
processingFilter.Add(new Dilatation());
processingFilter.Add(new Edges());
// apply the filter
Bitmap tmp1 = processingFilter.Apply(currentFrame);
```

After merging `tmp1 image` with the red channel of the original `image`, we'll get the following `image`:

![Image]

May be it looks not so perfect as the previous one, but the approach has a great possibility for performance optimization.

Looking at the previous picture, we can see, that objects are highlighted with a curve, which represents the moving object's boundary. But sometimes it's more likely to get a rectangle of the object. Not only this, what to do if we want not just highlight the objects, but get
their count, position, width and height? Recently I was thinking: "Hmmm, it's possible, but not so trivial". Don't be afraid, it's easy. It can be done using the BlobCounter class from my imaging library, which was developed recently. Using BlobCounter we can get the number of objects, their position and the dimension on a binary image. So, let's try to apply it. We'll apply it to the binary image containing moving objects, the result of Threshold filter.

```csharp
BlobCounter blobCounter = new BlobCounter();
...
// get object rectangles
blobCounter.ProcessImage(thresholdedImage);
Rectangle[] rects = blobCounter.GetObjectRectangles();
// create graphics object from initial image
Graphics g = Graphics.FromImage(image);
// draw each rectangle
using (Pen pen = new Pen(Color.Red, 1))
{
    foreach (Rectangle rc in rects)
    {
        g.DrawRectangle(pen, rc);
        if ((rc.Width > 15) && (rc.Height > 15))
        {
            // here we can highlight large objects with something else
        }
    }
}
g.Dispose();
```

Here is the result of this small piece of code. Looks pretty. Oh, I forgot. In my original implementation, there is some code instead of that comment for processing large objects. So, we can see a small numbers on the objects.
There was a lot of complains that the idea of `MoveTowards` filter, which is used for updating background `image`, is hard to understand. So, I was thinking a little bit about changing this filter to something else, which is clearer to understand. And the solution is to use `Morph` filter, which became available in 2.4 version of AForge.Imaging library. The new filter has two benefits:

- It is much more simpler to understand;
- The implementation of the filter is more efficient, so the filter produce better performance.

The idea of the filter is to preserve specified percentage of the source filter and to add missing percentage from overlay `image`. So, if the filter was applied to source `image` with percent value equal to 60%, then the result `image` will contain 60% of source `image` and 40% of overlay `image`. Applying the filter with percent values around 90% makes background `image` changing continuously to current frame.

**Motion Alarm**

It is pretty easy to add motion alarm feature to all these motion detection algorithms. Each algorithm calculates a binary `image` containing difference between current frame and the background one. So, the only we need is to just calculate the amount of white pixels on this difference `image`.

```csharp
// Calculate white pixels
private int CalculateWhitePixels( Bitmap image )
{
    int count = 0;
    // lock difference image
    BitmapData data = image.LockBits( new Rectangle( 0, 0, width, height ),
        ImageLockMode.ReadOnly, PixelFormat.Format8bppIndexed );
    int offset = data.Stride - width;
    unsafe
    {
        byte * ptr = (byte *) data.Scan0.ToPointer( );
        for ( int y = 0; y < height; y++ )
        {
            for ( int x = 0; x < width; x++, ptr++ )
            {
                count += ( (*ptr) >> 7 );
            }
            ptr += offset;
        }
        // unlock image
        image.UnlockBits( data );
    return count;
}
```

For some algorithms it could be done even simpler. For example, in blob counting approach we can accumulate not the white pixels count, but the area of each detected object. Then, if the computed amount of changes is greater than a predefined value, we can fire an alarm event.
Video Saving

There are many different ways to process motion alarm event: just draw a blinking rectangle around the video, or play sound to attract attention. But, of course, the most useful one is video saving on motion detection. In the demo application I was using the AVIWriter class, which uses Video for Windows interop to provide AVI files saving capabilities. Here is the small sample of using the class to write small AVI file, which draw diagonal line:

```csharp
SaveFileDialog sfd = new SaveFileDialog();
if (sfd.ShowDialog() == DialogResult.OK)
{
    AVIWriter writer = new AVIWriter("wmv3");
    try
    {
        writer.Open(sfd.FileName, 320, 240);
        Bitmap bmp = new Bitmap(320, 240, PixelFormat.Format24bppRgb);
        for (int i = 0; i < 100; i++)
        {
            bmp.SetPixel(i, i, Color.FromArgb(i, 0, 255 - i));
            writer.AddFrame(bmp);
        }
        bmp.Dispose();
    }
    catch (ApplicationException ex)
    {
    }
    writer.Dispose();
}
```

Note: In this small sample and in the demo application I was using Windows Media Video 9 VCM codec.

AForge.NET framework

The Motion Detection application is based on the AForge.NET framework, which provides all the filters and image processing routines used in this application. To get more information about the framework, you may read dedicated article on Code Project or visit project's home page, where you can get all the latest information about it, participate in a discussion group or submit issues or requests for enhancements.

Applications for motion detection

Some people ask me one question from time to time, which is a little bit strange to me. The question is "What is the application for motion detectors". There is a lot to do with them and it depends on the imagination. One of the most straightforward applications is video surveillance, but it is not the only one. Since the first release of this application, I've received many e-mails from different people, who applied this application to incredible things. Some of them have their own articles, so you can take a look:

- "Laser Gesture Recognition" by Ashish Derhgawen;
"Everyone Loves Babies! Webcams and Motion Detection" by Scott Hanselman;
"Motion-Detecting, Blood Squirtling Halloween Skull" by Brian Peek.

Conclusion

I’ve described only ideas here. To use these ideas in real applications, you need to optimize its realization. I’ve used an image processing library for simplicity, it’s not a video processing library. Besides, the library allows me to research different areas more quickly, than to write optimized solutions from the beginning. A small sample of optimization can be found in the sources.

History

- [20.04.2007] - 1.5
  - Project converted to .NET 2.0;
  - Integrated with AForge.NET framework;
  - Motion detectors updated to use new features of AForge.NET to speed-up processing.
- [15.06.2006] - 1.4 - Added fifth method based of Morph filter of AForge.Imaging library.
- [08.04.2006] - 1.3 - Motion alarm and video saving.
- [22.08.2005] - 1.2 - Added fourth method (getting objects’ rectangles with blob counter).
- [01.06.2005] - 1.1 - Added support of local capture devices and MMS streams.

About the Author

Andrew Kirillov

Started software development at about 15 years old and it seems like now it lasts most part of my life. Fortunately did not spend too much time with Z80 and BK0010 and switched to 8086 and further. Similar with programming languages – luckily managed to get away from BASIC and Pascal to things like Assembler, C, C++ and then C#. Apart from daily programming for food, do it also for hobby, where mostly enjoy areas like Computer Vision, Robotics and AI. This led to some open source stuff like AForge.NET.

Going out of computers I am just a man loving his family, enjoying traveling, a bit of books, a bit of movies and a mixture of everything else. Always wanted to learn playing guitar, but it seems like 6 strings are much harder than few dozens of keyboard’s keys. Will keep progressing ...