Design of Multimedia Applications

Part 2: Video Shot Detection, Annotation, and Retrieval

October 25, 2013

1 Introduction

Easy-to-use multimedia devices and services, cheap storage and bandwidth, and an increasing number of people going online have resulted in a surge of the *consumption* of online video content. As an example, people currently watch over six billion hours of video each month on YouTube [1]. Similarly, the digitization of professional video archives and the popularity of user-generated video content have resulted in a surge of the *availability* of online video content. Again using YouTube as an example, people currently upload 100 hours of video content every minute to the aforementioned website.

The above statistics demonstrate that online video content has become highly popular. However, these statistics are also indicative of the problem of digital video overload: given the time-consuming and subjective nature of manual video annotation, our ability to organize video content is not able to keep up with our ability to create and store video content. This makes it for instance more and more difficult for users to find video content that is in line with their interests.

Given the pressing need for computerized techniques that allow for effective and efficient organization of vast amounts of video content, automatic video content understanding is currently an area of intense research and development. Since identification of the temporal structure of video content is essential to video content understanding, shot boundary detection is commonly the first step taken, followed by the selection of representative frames and the subsequent usage of machine learning techniques for detecting semantic concepts like 'face', 'sunset', 'beach', 'building', and so on [2].

This lab session aims at giving a better insight into the problem of (large-scale) video content understanding. To that end, we will implement and evaluate a number of state-of-the-art techniques for shot boundary detection, and we will subsequently integrate these techniques into a DirectShow-based application that offers support for the manual annotation and retrieval of video shots.

2 Video shot detection

2.1 Scene versus shot

In the literature, the detection of shots is also called scene detection, unfortunately mixing up the concepts *scene* and *shot*. To make a clear distinction, we define these two concepts as follows:

Shot: a sequence of consecutive frames produced by a single camera.

Scene: a sequence of consecutive shots that are semantically connected, i.e. the shots have a common topic or the shots have been filmed at the same location.

Figure 1 shows the temporal structure typically present in a video sequence.

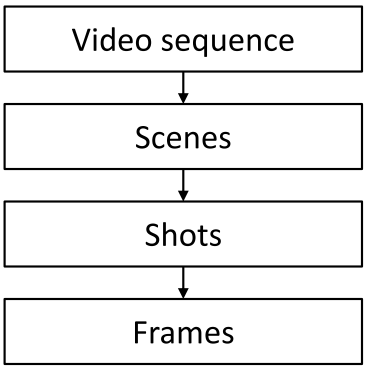


Figure 1: Temporal structure of a video sequence.

2.2 Video shots

Shots can be present in a video sequence in different ways.

Cut: an *abrupt transition* from one shot to the next. Traditionally, movies were cut in pieces and pasted back together, as such yielding the term 'Cut'. An example of a cut can be found in Figure 2.

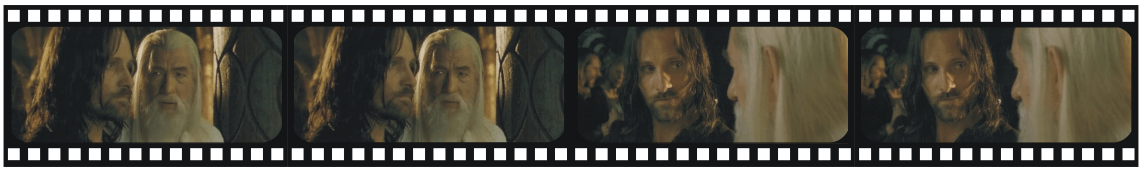


Figure 2: Example of a cut.

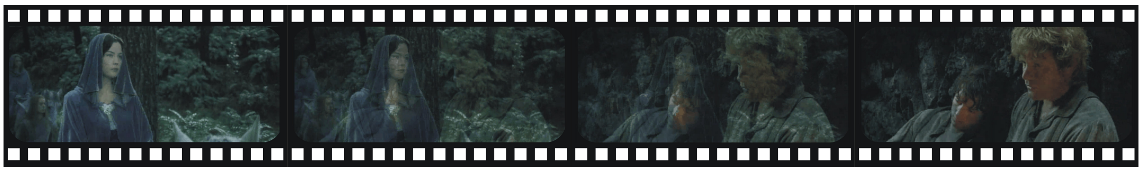


Figure 3: Example of a dissolve.

Dissolve: a *gradual transition* during which a shot fades into a new shot. An example of a dissolve can be found in Figure 3.

Fade: a special kind of *dissolve*, with a shot dissolving into a still image (usually uniformly colored), or vice versa. If a fade starts from a still image and ends with a shot, then the fade is called a *fade-in*. If a fade starts from a shot and ends with a still image, then the fade is called a *fade-out*. It should be clear that fades are part of the family of *gradual transitions*.

An example of a fade-in can be found in Figure 4.

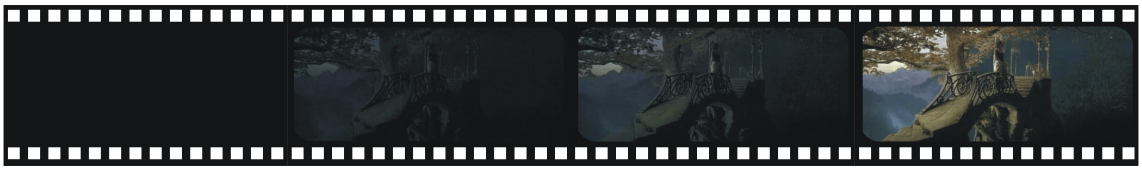


Figure 4: Example of a fade-in.

An example of the reverse case, a fade-out, can be found in Figure 5.

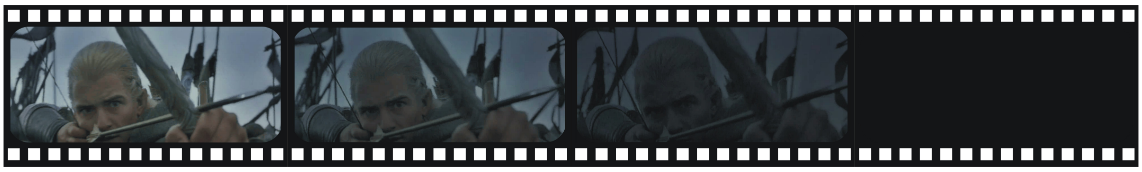


Figure 5: Example of a fade-out.

3 Methods for video shot detection

This section briefly reviews a number of common methods for detecting shots in video sequences. Note that the methods discussed operate in the pixel domain (this is, the uncompressed or spatial domain), meaning that these methods directly analyse pixel values. During the past few years, a number of methods have also been proposed that are able to operate in the compressed domain [3]. Further, note that the methods discussed focus on the detection of cuts. However, by keeping track of multiple consecutive frame differences, the methods discussed should also be able to deal with gradual shot transitions.

3.1 Pixel comparison

The easiest way to detect differences between two frames is to calculate the pixel-wise differences. This technique is generally referred to as *pixel difference*. A straightforward way to calculate pixel-wise differences is to make use of the formula that has been proposed by Nagaska and Tanaka [4], as shown in the equation below:

$$D(i,i+1) = \frac{\sum_{x=1}^{X}\sum_{y=1}^{Y}|P_i(x,y) - P_{i+1}(x,y)|}{X \times Y}. \tag{1}$$

$P_i(x,y)$ denotes the intensity value of a pixel with coordinates x and y in frame i (with i denoting the frame number). X and Y represent the dimensions of the frame[1]. It should be clear that the pixel-wise differences are averaged over the entire frame.

We can extend the formula to color frames as follows:

$$D(i,i+1) = \frac{\sum_{x=1}^{X}\sum_{y=1}^{Y}\sum_{c}|P_i(x,y,c) - P_{i+1}(x,y,c)|}{X \times Y}. \tag{2}$$

The parameter c denotes the index of the color components (e.g., $c \in \{R,G,B\}$ or $c \in \{Y,Cb,Cr\}$). Consequently, $P_i(x,y,c)$ holds the value of the color component c of the pixel with coordinates x and y in frame i. Figure 6 illustrates this calculation for one pixel of a particular frame. In the example, the difference is calculated for the blue component of the indicated pixel location.

Using a *threshold* δ_1, we are able to detect those frames in the video sequence that exhibit a large difference with the next frame:

[1]We can assume that both frames have the same dimension since they are part of the same video sequence.

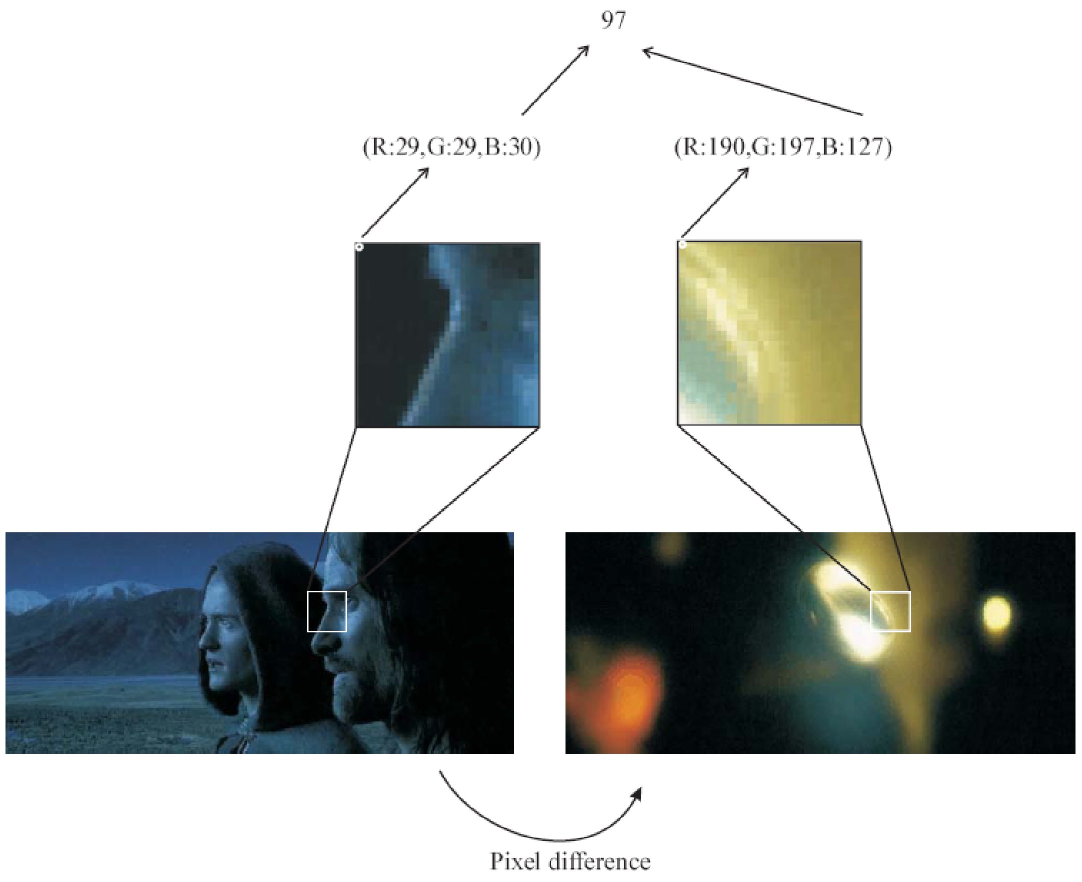


Figure 6: Illustration of the *pixel difference* method.

$$D(i, i+1) > \delta_1. \tag{3}$$

Note that this method only compares two consecutive frames at a time. Therefore, this method is not suited for detecting gradual transitions.

In order to make shot detection more robust against object and/or camera motion, Zhang *et al.* proposed to extend the method of Nagaska and Tanaka by including only large pixel differences in the total sum [5]. This is shown in Equation 4. The difference of two pixels is now first compared with a threshold δ_2. Consequently, only large differences are used to find the final value for $D(i, i+1)$, as shown in Equation 5.

$$DP(i, i+1, x, y) = \begin{cases} 1 & \text{if } \sum_c |P_i(x,y,c) - P_{i+1}(x,y,c)| > \delta_2 \\ 0 & \text{else} \end{cases} \tag{4}$$

$$D(i, i+1) = \frac{\sum_{x=1}^{X} \sum_{y=1}^{Y} DP(i, i+1, x, y)}{X \times Y} \tag{5}$$

Finally, a threshold δ_3 can be applied to the frame difference computed in Equation 5.

3.2 Block comparison

If two frames are compared pixel-wise, then only global characteristics of the frames are used. To take advantage of local (spatial) frame characteristics, *block-based algorithms* have been proposed in the scientific literature.

Block-based algorithms are less sensitive to small movements by the camera or objects in the video sequence. Each frame is divided into a number of non-overlapping blocks. These blocks are then compared to the blocks at corresponding positions in the following frame. The block difference can be expressed as follows:

$$D(i, i+1) = \sum_{k=1}^{b} c_k \cdot DP(i, i+1, k), \tag{6}$$

where i denotes the frame number, b represents the number of blocks in a frame, c_k is a weight given to each block, and $DP(i, i+1, k)$ is the block difference.

Kasturi *et al.* used a probability equation to compare two blocks [6]:

$$\lambda_k = \frac{\left[\frac{\sigma_{k,i}+\sigma_{k,i+1}}{2} + \left(\frac{\mu_{k,i}-\mu_{k,i+1}}{2}\right)^2\right]^2}{\sigma_{k,i} \cdot \sigma_{k,i+1}}, \tag{7}$$

where $\mu_{k,i}$ and $\mu_{k,i+1}$ represent the average intensity values of block k in frame i and $i+1$, respectively. Accordingly, $\sigma_{k,i}$ and $\sigma_{k,i+1}$ denote the standard deviation of these blocks. The number of blocks for which λ_k is larger than a threshold δ_4 is consequently represented by $D(i, i+1)$ (see Eq. 6), with $DP(i, i+1, k)$ given by:

$$DP(i, i+1, k) = \begin{cases} 1 & \text{if } \lambda_k > \delta_4 \\ 0 & \text{else} \end{cases} \tag{8}$$

If $D(i, i+1)$ is larger than a second threshold δ_5, then a cut is detected. This method is visualized in Figure 7.

This method is less influenced by small movements between two consecutive frames. However, this method is computationally more complex.

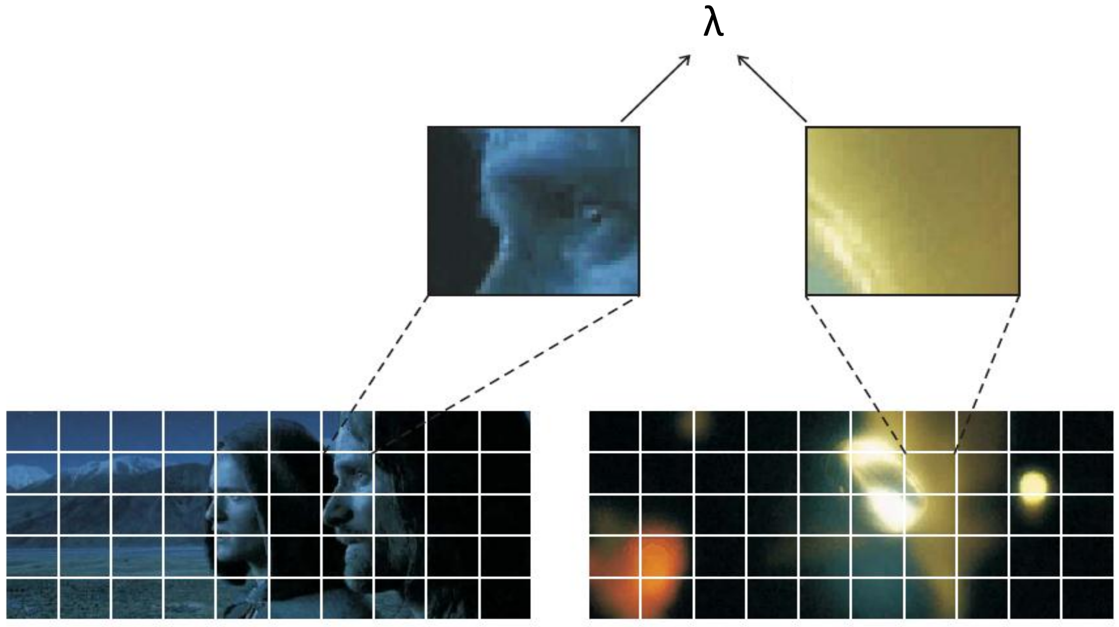


Figure 7: Illustration of a *block-based comparison* method.

3.3 Motion estimation

It should be clear that the two previous techniques for shot detection have problems when a video sequence contains a lot of motion. Hence, a method that analyzes the motion between two consecutive frames will likely yield better results.

During motion estimation, a frame is typically divided into (square) blocks in order to find blocks in the next frame that resemble the blocks in the frame under consideration. To decrease the computational complexity, a search window is used to find the best matching block. As such, for each block b in frame i, we look for the best matching block p within a search window (centered around the position of block b) in frame $i+1$. The best matching block p minimizes the block difference $D_b(b_1, b_2)$ given by the following equation:

$$D_b(b_1, b_2) = \sum_{x=1}^{X_b} \sum_{y=1}^{Y_b} |P_{b_1}(x, y) - P_{b_2}(x, y)|. \tag{9}$$

In this equation, X_b and Y_b denote the width and height of the blocks, and $P_{b_1}(x, y)$ and $P_{b_2}(x, y)$ denote the value of the pixel with coordinates x and y in block b_1 and in block b_2, respectively.

Once the best matching block p_k in frame $i+1$ has been found for each block k in the current frame i, we can calculate the image difference as follows:

$$D(i, i+1) = \sum_{k=1}^{b} D_b(k, p_k). \tag{10}$$

If good matches have been found for many blocks, we can assume that the frame content did not change much. In contrast, if a cut occurs, it is likely that no good match can be found for many blocks. By comparing $D(i, i+1)$ with a threshold δ_6, a cut can be detected.

3.4 Histogram comparison

A histogram of a frame i is an n-dimensional vector $H_i(j)$ with $j = 1, \ldots, n$ and with n denoting the number of gray or color values. Each element of this vector holds the number of pixels that have the corresponding gray or color value.

Figure 8 shows an example histogram of gray values for the frame shown. The histogram shows, for each gray value, the number of pixels in the frame that have that value. We can observe that a local maximum is present between values 50 and 60. These correspond to the color of the background in the frame. The gray values around 225 also show an increase in the number of pixels that have those values. These pixels correspond to the bright text in the frame[2].

In this section, we will only discuss gray-scale histograms.

3.4.1 Global histogram comparison

Once the histograms of two consecutive frames have been created, we can compare these in order to detect a cut. The following equation can be used to calculate the difference between two histograms:

$$D(i, i+1) = \sum_{j=1}^{n} |H_i(j) - H_{i+1}(j)|. \tag{11}$$

$H_i(j)$ denotes the histogram value of the gray value j of frame i, whereas n denotes the total number of different gray values.

A way to reduce n is to create bins that correspond to several gray values. For example, if we assign 16 consecutive gray values to one bin, we can represent the entire range of gray values by 16 bins. The histogram value for a certain bin then corresponds to the number

[2]A gray value of 0 denotes a black pixel, whereas a value of 255 denotes a white pixel.

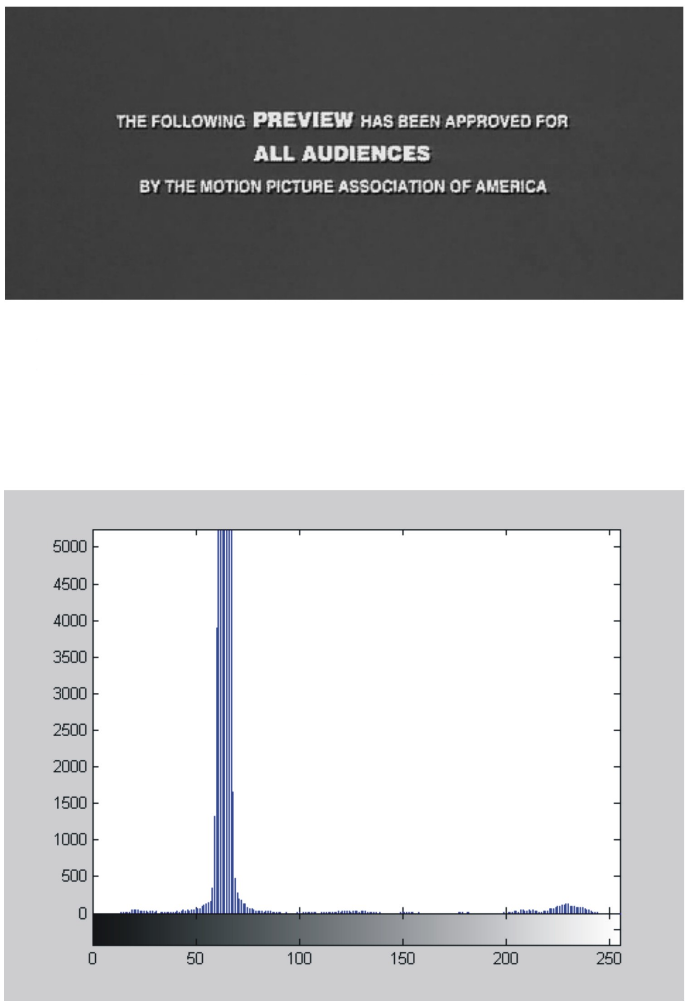


Figure 8: Illustration of a histogram of gray values. Note that the histogram can be made independent of the spatial resolution by dividing the number of pixels in the frame that have a certain gray value by the total number of frame pixels (normalization).

of pixels that have a gray value that falls in the range of the bin under consideration. This uniform partitioning of the range of possible gray values, which is also known as fixed binning, allows for a significant reduction of the computational complexity. Note that, dependent on the use case targeted, a content-adaptive partitioning can be used as well (so-called adaptive binning; [7]).

Again, a threshold can be used to evaluate the frame difference computed by means of the aforementioned histograms.

3.4.2 Local histogram comparison

A global histogram comparison is more robust against camera or object motion. However, since the histograms are calculated for entire frames, they cannot detect changes for which the distribution of gray values does not change much. For example, the different frames in Figure 9 all yield identical histograms.

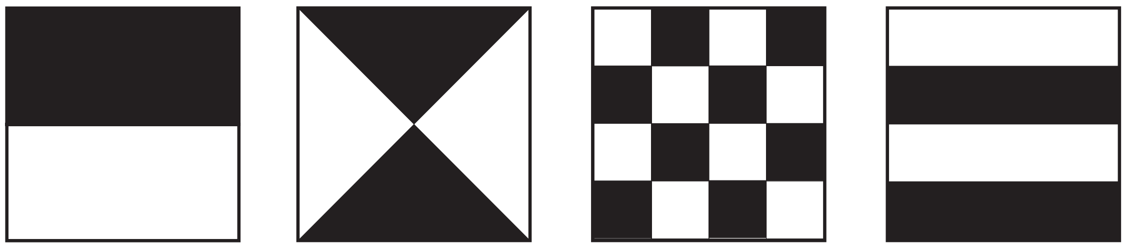


Figure 9: Illustration of frames that all yield identical histograms.

To prevent this, *local histogram comparison* can be used. Yihong *et al.* [8] proposed to divide frames in nine regions or blocks. For each of these blocks, the histogram is calculated and then used for comparison purposes. This approach is in fact a block-based histogram comparison. The difference between two consecutive frames can then be found by Equation 12.

$$\begin{aligned} D(i, i+1) &= \textstyle\sum_{k=1}^{b} DP(i, i+1, k) \\ DP(i, i+1, k) &= \textstyle\sum_{j=1}^{n} |H_i(j, k) - H_{i+1}(j, k)| \end{aligned} \tag{12}$$

$H_i(j, k)$ represents the histogram value of gray value j in block k, with b the total number of blocks.

Note that the use of local histograms as discriminative feature vectors is currently highly popular in the area of computer vision and pattern recognition, as well as in the area of content-based image and video retrieval. Such histograms are often computed for luma regions that are surrounding points-of-interest in still images or key frames [9].

4 Effectiveness of video shot detection

To evaluate the effectiveness of a particular method for shot detection, the output of the shot detection method is compared to a ground truth, and where the output of the shot detection method states at which frame a shot transition starts. A manual ground truth can be made by simply going through the video sequence on a frame-by-frame basis and identifying each shot transition.

For the evaluation, we use concepts from the field of statistics. If a particular method correctly detects a shot transition, then this is called a true positive (TP). In contrast, if the method detects a shot transition at a certain frame, but a shot transition does not occur in reality, then this is called a false positive (FP). If the method misses a shot transition, then this is called a false negative (FN).

A first measure is the **recall**, which is defined as the ratio between the true positives and the sum of the true positives and the false negatives:

$$recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}. \tag{13}$$

As an example, if there are 100 shots in a particular sequence and 80 of them have been correctly detected by the algorithm, then the recall will be 80%.

The **precision** denotes the percentage of all shots detected that actually correspond to real shots:

$$precision = \frac{True\ Positives}{True\ Positives + False\ Positives}. \tag{14}$$

To calculate the precision and recall values, the output of shot detection is compared to a ground truth. It should be clear that effective methods for shot detection need to obtain high precision and recall values. In addition, the combination of precision and recall is necessary to correctly interpret results. For example, a sensitive method for shot detection may find a shot every two frames, leading to a high recall (many of the actual shots are detected), but to a low precision (only a small percentage of the detected shots correspond to real shots).

5 DirectShow

DirectShow, which is short for the Microsoft DirectShow application programming interface (API), is a popular streaming media architecture for Microsoft Windows. Using DirectShow, applications are able to capture and playback audio and video. Initially, DirectShow was included in the DirectX SDK. The latest version of the DirectX SDK to include DirectShow was DirectX 9.0 SDK Update - (February 2005) Extras[3]. After this version, DirectShow was moved to the Windows SDK.

DirectShow supports a wide variety of coding and storage formats, including Advanced Systems Format (ASF), Moving Picture Experts Group (MPEG), Audio-Video Interleaved (AVI), MPEG Audio Layer-3 (MP3), and WAV sound files. It supports capture from digital and analog devices based on the Windows Driver Model (WDM) or Video for Windows (VfW). It automatically detects and uses video and audio acceleration hardware when available, but also supports systems without acceleration hardware.

Internally, DirectShow works with filter graphs that may consist of different types of filters and that are managed by the Filter Graph Manager. In what follows, we provide explanatory notes about the *Filter Graph Manager* and the three types of filters available in DirectShow.

5.1 The filter graph

A *filter graph* is a collection of filters that are interconnected to provide a certain functionality. This can be best illustrated using the *GraphEdit* tool. This tool allows to easily create filter graphs, as shown in Figure 10.

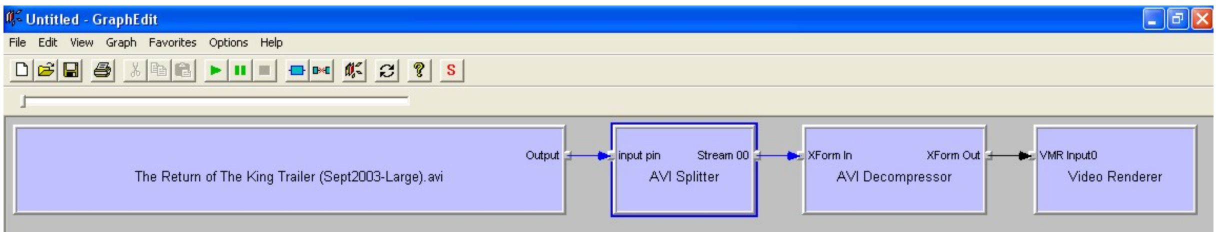


Figure 10: Screenshot of GraphEdit.

This filter graph allows playing a video sequence. Besides the filters, Figure 10 shows the connections that hold the audio and video streams.

Filter graphs are highly similar to the concept of mathematical graphs. The filters are the nodes of the graph, connected by means of edges. Some filters can only serve as start nodes of the graph (i.e., source filters). Other filters can only serve as end nodes of the graph (i.e., renderer filters). Finally, the remaining nodes constitute a third type of filters (i.e., transform filters).

[3]For more information, please visit `http://www.microsoft.com/windows/directx/`.

Note that the filter graph maintains a clock in software to synchronize the filters. This is necessary because the different types of *media samples* need to be kept in sync. For example, audio and video streams should be running synchronously.

5.2 Filters

A filter is the basic entity of DirectShow. The filters contain *pins* that either input or output data. Consequently, they are called *input pins* or *output pins*. Not every output pin can be connected to every input pin. Each pin only accepts certain media formats and to connect pins, compatible formats need to be used.

In DirectShow, three types of filters can be found.

Source filter: The source filter produces a data stream and only has output pins. The stream of data can originate from a multimedia file on the hard disk, or from a device capable of registering live data (e.g., a web cam or a microphone). In Figure 10, the source filter can be seen on the left.

Renderer filter: While the source filter forms the starting point of the data flow in a filter graph, the renderer filter is the end point. Filters of this type, which only have input pins, make sure that the multimedia stream reaches its destination. This allows writing data to a file, visualizing these data on a screen, or sending these data to speakers.

Transform filter: This type of filter has both input and output pins. The data stream flowing through the filter may be the subject of a transformation.

Two approaches exist to connect the pins of a pair of filters. They can be manually connected or in an automated manner, using *Intelligent Connect*. This mechanism automatically inserts the necessary transform filters in order to allow sending a stream between the filters that need to be connected.

5.3 A DirectShow application

DirectShow is designed for C++. Microsoft does not provide a managed API for DirectShow. However, such an API has been made available by a third party, and is known as the DirectShow.net library (`http://directshownet.sourceforge.net/`).

The purpose of this library is to allow access to the DirectShow functionality of Microsoft Windows from within .NET applications. This library supports both Visual Basic .NET and C#, and theoretically, it should work with any .NET language.

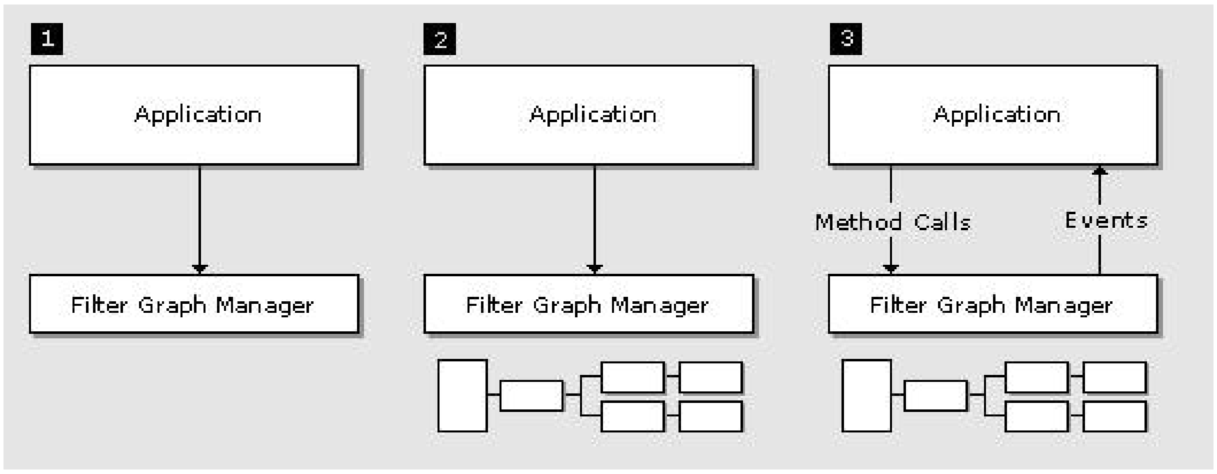
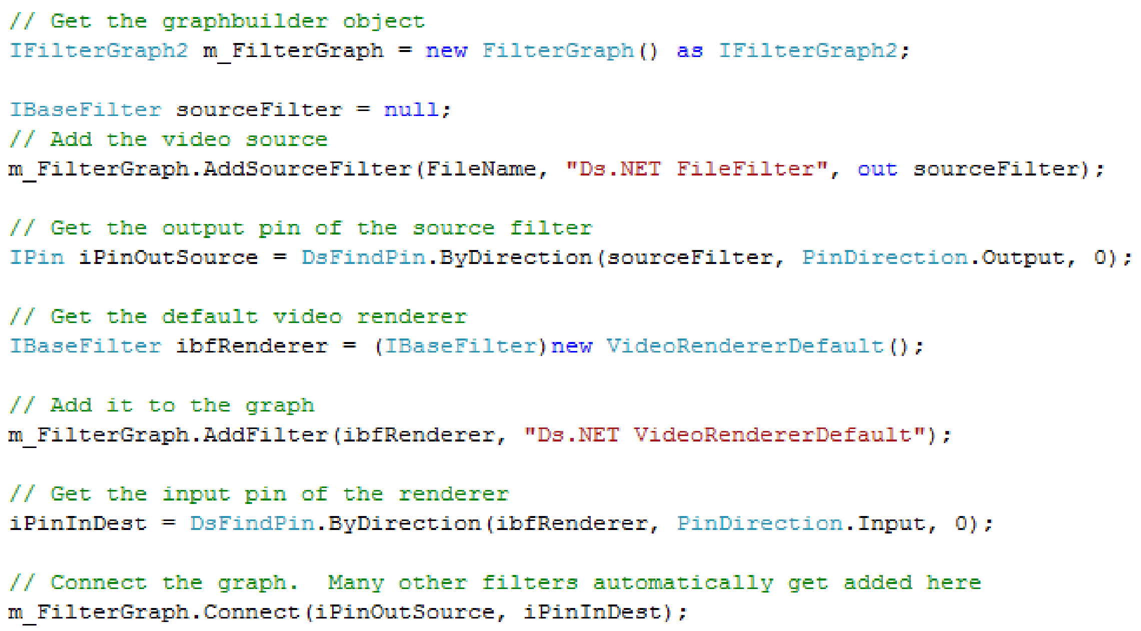


Figure 11: Screenshot of a DirectShow application.

In broad terms, there are three tasks that any DirectShow application must perform. These tasks are illustrated in the diagram in Figure 11. Explanatory notes can be found below:

1. The application creates an instance of the Filter Graph Manager.
2. The application uses the Filter Graph Manager to build a filter graph. The exact set of filters in the graph will depend on the application.
3. The application uses the Filter Graph Manager to control the filter graph and stream data through the filters. Throughout this process, the application will also respond to events from the Filter Graph Manager.

An example of such a filter graph is shown in Figure 12. First, a filter graph object is created. Next, a source filter is created that opens a file corresponding to *FileName*. Since this is a source filter, it has only output pins. We can retrieve the output pin for this filter by using the *DsFindPin.ByDirection* method. Next, a renderer filter is created. To that end, we make use of the default video renderer. This filter is consequently added to the filter graph and the input pin of the filter is retrieved. Finally, the output pin of the source filter and the input pin of the renderer filter are connected. DirectShow will automatically add the necessary filters to make sure that this connection can be established.



```
// Get the graphbuilder object
IFilterGraph2 m_FilterGraph = new FilterGraph() as IFilterGraph2;

IBaseFilter sourceFilter = null;
// Add the video source
m_FilterGraph.AddSourceFilter(FileName, "Ds.NET FileFilter", out sourceFilter);

// Get the output pin of the source filter
IPin iPinOutSource = DsFindPin.ByDirection(sourceFilter, PinDirection.Output, 0);

// Get the default video renderer
IBaseFilter ibfRenderer = (IBaseFilter)new VideoRendererDefault();

// Add it to the graph
m_FilterGraph.AddFilter(ibfRenderer, "Ds.NET VideoRendererDefault");

// Get the input pin of the renderer
iPinInDest = DsFindPin.ByDirection(ibfRenderer, PinDirection.Input, 0);

// Connect the graph.  Many other filters automatically get added here
m_FilterGraph.Connect(iPinOutSource, iPinInDest);
```

Figure 12: Example code for a Directshow application rendering video.

6 Exercises

6.1 Instructions

The following files need to be uploaded to Minerva, and in particular to the Dropbox module:

1. All software, including the project and solution files:
   `shot_detection_src_<group number>.zip`.
2. The report, in PDF or MS Word format:
   `shot_detection_report_<group number>.pdf` or
   `shot_detection_report_<group number>.doc(x)`.

Deadlines

- **5 December 2013, 16h00 (Thursday)**: software.

- **12 December 2013, 16h00 (Thursday)**: report.

Remarks

- Only the files requested need to be uploaded to Minerva.
- Please make sure that all files have been named correctly, and that all files have been uploaded to the Dropbox on Minerva.
- Grading takes into account the structure (e.g., usage of helper functions), readability, and documentation of the code written. **Note that code needs to be documented in English!**

6.2 Background information

In this lab session, we will implement and evaluate the effectiveness and efficiency of a number of the shot detection algorithms discussed. These algorithms will be embedded in a Windows Forms C# Application. Note that this lab session targets both algorithmic development and application development.

6.3 Application

The goal is to create an application for video shot detection, annotation, and retrieval that offers its functionality to users by means of a Graphical User Interface (GUI). The application needs to be designed as a Windows Forms C# application. In order to manipulate video sequences, we will make use of the DirectShow.net library. For this series of exercises, a number of examples available on Minerva can be used as a starting point (DxPlay, DxScan, DxText).

The application **must** provide the following functionality:

open video Select a video sequence using the file system.

play video Play the original video sequence.

detect shots Identify shots by means of different methods for video shot detection (see Section 6.4). It should be possible to change the parameters of these methods. In addition, the meaning of the different parameters, as well as their range, should be clear from the GUI.

show shots Visualize the different shots in the video sequence (e.g., by making use of representative frames and/or frame numbers).

play shot Play a specific video shot.

annotate shot Manually annotate a shot with one or more keywords (tags). Store the keywords in the XML file containing the shot information. The way the keywords are stored in the XML file is at your discretion.

export shot info Export the shot information as an XML file (see Section 6.5).

export frames Export a representative frame per shot as a still image to the file system.

retrieve shots Retrieve and show all shots that have been annotated with a particular keyword (e.g., return all shots that have been tagged with 'people').

Remarks

- More sample applications can be found at `http://directshownet.sourceforge.net/index.html`.
- Be sure to download the DirectShow library and to change the relevant references in the example projects.

- To add a library to a project, right click the ‘References’ folder. Then choose ‘Add Reference...’. Select the tab ‘Browse’ and insert the library.
- The use of different codec versions (e.g., the use of ffdshow on different PC configurations – ‘codec hell’) may result in decoded frames that have pixels with slightly different color values.
- GraphEdit allows visualizing the filter graph used to render a media file.
- Extensive DirectShow documentation can be found at `http://msdn.microsoft.com/en-us/library/dd375454(VS.85).aspx`.

6.4 Algorithms

This section describes the different video shot detection techniques that need to be implemented. Note that the first four exercises aim at the detection of cuts, whereas the last exercise aims at the detection of both cuts and gradual shot transitions. In order to evaluate the first four exercises, the Star Wars video sequence needs to be used (this sequence only contains cuts). The last exercise needs to be evaluated by means of Star Wars, Youth Without Youth, and CSI (both the Youth Without Youth and the CSI video sequence contain a high number of gradual shot transitions). The video sequences and corresponding ground truths (see Section 6.5) can be downloaded from Minerva.

6.4.1 Excercise 1: pixel difference method for detecting abrupt shot transitions

Implement the method `PixelDifferenceSD`. To that end, make use of the pixel difference method proposed by Zang *et al.*. The two parameters of this method are discussed in Section 3.1. Provide support for timing the execution of this method.

6.4.2 Excercise 2: motion estimation method for detecting abrupt shot transitions

Implement the method `MotionEstimationSD`. To that end, make use of motion estimation, as discussed in Section 3.3. The parameters used by this method are the threshold to evaluate the image difference, the block size used, and the size of the search window (i.e., the distance of the center of the search window to the boundary of the search window in terms of pixels, in other words, half the width or half the height of the search window). Provide support for timing the execution of this method.

6.4.3 Excercise 3: global histogram method for detecting abrupt shot transitions

Implement the method `GlobalHistogramSD`. To that end, make use of global histograms **for color images** and fixed binning, as discussed in Section 3.4.1. The parameters are the threshold and the number of bins. Provide support for timing the execution of this method.

6.4.4 Excercise 4: local histogram method for detecting abrupt shot transitions

Implement the method `LocalHistogramSD`. To that end, make use of local histograms **for color images** and fixed binning, as discussed in Section 3.4.1. The parameters are the threshold, the number of bins, and the region size (in terms of pixels). Provide support for timing the execution of this method.

6.4.5 Excercise 5: generalized method for detecting abrupt and gradual transitions

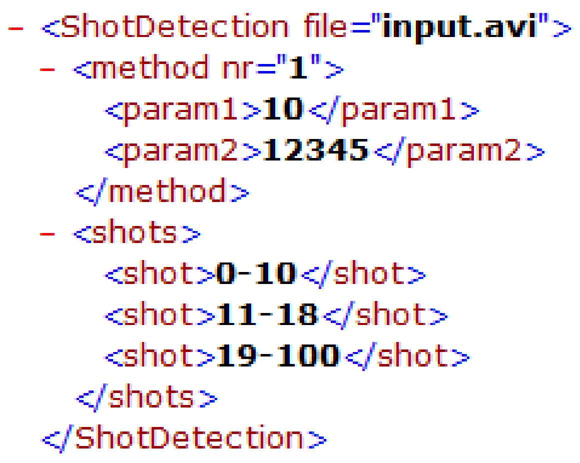
Implement the method `GeneralizedSD`. Either extend one of the previously implemented methods for video shot detection with support for the identification of gradual transitions, or implement a method found in the scientific literature (in the latter case, a reference needs to be provided to the paper discussing the method implemented). How to deal with the different problems encountered is at your discretion. More advanced methods will result in a better shot detection performance and will also be graded higher.

6.5 Evaluation and report

6.5.1 Computation of precision and recall

Quantifying the effectiveness of a shot detection algorithm, as well as many other computer vision and pattern recognition algorithms, can be done by calculating precision and recall values. To that end, a ground truth needs to be available for each video sequence in the training set, describing the shots that are truly present in the training video sequences. An XML-based ground truth for the video sequences used in this lab session can be found on Minerva (this is, for the Star Wars, the Youth Without Youth, and the CSI video sequences). The application for shot detection should make it possible to export the shot detection results in a similar XML format. The XML file created should resemble the output depicted in Figure 13:

1. The method used needs to be listed (the numbering of the methods should follow the ordering of the exercises provided in this document).
2. The values of the parameters used need to be supplied.
3. Each shot detected needs to be represented by its first and last frame (frames are counted starting from zero).



```
- <ShotDetection file="input.avi">
  - <method nr="1">
      <param1>10</param1>
      <param2>12345</param2>
    </method>
  - <shots>
      <shot>0-10</shot>
      <shot>11-18</shot>
      <shot>19-100</shot>
    </shots>
  </ShotDetection>
```

Figure 13: Example of the XML output.

Note that the .Net framework provided includes support for handling XML files (through the *XMLDocument* class).

6.5.2 Exercise 6: Evaluation and report

To evaluate the effectiveness of the different shot detection techniques implemented, the precision and recall values need to be computed for the video sequences available on Minerva. You can make use of the ground truth files and the output of your algorithms in order to automatically compute the precision and recall values for the video sequences used. Note that you may assume that precision and recall have the same importance. Also, note that, if the parameters of an algorithm change, the output will change accordingly. Consequently, different precision and recall values will be obtained for different parameter settings. It is not necessary to deal with all possible parameter settings; restrict the settings used to a meaningful subset, and where the parameter settings used need to be justified in the report submitted (see further).

Finally, the results obtained need to be discussed in a clear report (written in English), containing:

1. Pseudo-code and explanatory notes for the video shot detection method implemented in Exercise 5.
2. A table documenting the different parameters used, as well as explanatory notes regarding the *definition* and the *range* of these different parameters. Also, add a rationale for the different parameter settings used (this is, how did you restrict the parameter settings to a meaningful subset?).
3. A comparative discussion of the precision and recall values obtained for each video shot detection method, taking into account different parameter settings.
4. A comparative discussion of the computational complexity of each video shot detection method, taking into account different parameter settings. Provide information about the PC configuration used in order to put the time measurements in context.
5. A summarizing ROC curve for each video sequence, illustrating the effectiveness of the different techniques implemented (see slide 26 of the introductory lecture). Corresponding tables need to be provided as well, containing the optimal parameter settings and the resulting precision, recall, and F_1 score (the harmonic mean of precision and recall).
6. A discussion of the differences between the video shot detection methods, and the influence of these differences on 1) the precision and recall values and 2) the computational complexity.
7. An answer, with explanation, to the following questions:
   - Can a clear winner be found among all different video shot detection methods and parameter settings?
   - For the clear winner found, or for the video shot detection method that you would use in practice:
     - How would you further improve this method in terms of precision?
     - How would you further improve this method in terms of processing speed?

- What is the obvious shortcoming in practical scenarios, that the different video shot detection methods have in common, and how would you deal with this?
- Given a video sequence produced by a wearable computing device like Google Glass (a/o), what issues does a method for video shot detection need to overcome?
- Why is it important to add metadata to video content?
- Given the annotation and retrieval functionality of the application written, what are some of the (engineering/user) problems that you foresee when you would deploy this application in the real world?

Make sure that the report is well-structured and well-written, and that the report answers all of the questions listed above. Pay attention to the proper use of visualizations and summarizing ROC plots (e.g., labeling of axes). Note that the use of bullets may be helpful in structuring the answers to the questions listed above.

The number of pages is limited to six (a seventh page will not be read!). So, keep the report focused and concise. Note that supportive screenshots can be added as an appendix (e.g., a screenshot of the GUI), and that this appendix does not add to the page count.

References

[1] YouTube. YouTube Statistics. Available on `http://www.youtube.com/t/press_statistics/` (Last accessed: October 15, 2013), 2013.

[2] Cees G.M. Snoek and Arnold W.M. Smeulders. Visual-Concept Search Solved? *IEEE Computer*, 43(6):76–78, 2010.

[3] Sarah De Bruyne, Davy Van Deursen, Jan De Cock, Wesley De Neve, Peter Lambert, and Rik Van de Walle. A compressed-domain approach for shot boundary detection on H.264/AVC bit streams. *Signal Processing: Image Communication*, 23(7):473–489, 2008.

[4] A. Nagaska and Y. Tanaka. Automatic Video Indexing and Full-Video Search for Object Appearances. *Journal of Information Processing*, 15(2), 1992.

[5] H. Zhang, A. Kankanhalli, and S. Smoliar. Automatic Partitioning of Full-Motion Video. *ACM Multimedia Systems Journal*, 1(1):10–28, 1993.

[6] R. Kasturi and R. Jain. Dynamic Vision. *Computer Vision: Principles*, 1991.

[7] Wee Kheng Leow and Rui Li. The analysis and applications of adaptive-binning color histograms. *Computer Vision and Image Understanding*, 94(1-3):6791, 2004.

[8] G. Yihong, Z. Hongjiang, and C.H. Chuan. An Image Database System with Fast Image Indexing Capability based on Color Histograms. *IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology*, 1:94–106, 1994.

[9] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.