UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

Software Engineering for Interactive Media

Labs Report

Instructor : Vincent CHARVILLAT

[1] *University of Toulouse*

[2] *University of Science and Technology of Hanoi*

Student : VU Anh Tuan

Master ICT 13-15, USTH

Hanoi, June 14, 2014

Contents

List of Figures

List of Codes

1 User Studies

The goal of this section is to introduce basically the concept of **user studies**. Some definitions are presented in the section 1. The section 2 is reserved to discuss some reasons that need to conduct user studies. At last, in section 3, I will talk about some limitations of user studies.

1. Definitions

There are many definitions of user studies. In this section, I will present some definitions that are the most appropriate in my opinion.

1. User studies are research studies focusing on the use of information sources from "real" (end) users. [2]
2. User studies are extended on the channels of information, such as means (formats), networks (social, academic, workplace networks) or information providing systems (information retrieval systems, digital libraries, web search systems, etc.). [2]
3. In user studies, the use of information sources in considered the dependent variable, while the various aspects of the users' profile are considered the independent variable. [2]
4. User studies offer a scientifically sound method to measure a visualization's performance. [3]

2. Why conduct user studies?

Visualization is mostly a craft. Methods are often designed and evaluated by presenting results informally to potential users. No matter how efficient a visualization technique may be, or how well motivated from theory, if it does not convey information effectively. Therefore, we need to conduct user studies. This is only one of reasons to carry out user studies. There still exist many other reasons to pursue user studies.

- Studies can be used to evaluate the strengths and weaknesses of different visualization techniques.

- Studies can show that a new visualization technique is useful in a practical sense, according to some objective criteria, for some specific task.
  Even more exciting are studies that show that a new technique is more effective than an existing technique for an important task, where the existing technique was previously considered the *best* technique to use.

- User studies can objectively establish which method is most appropriate for a given situation.

- A more fundamental goal of conducting user studies is to seek insight into *why* a particular technique is effective.
  This can guide future efforts to improve existing techniques. We want to understand for what types of tasks, and under what conditions, a particular method will give high quality results. This knowledge is critical, since different analysis tasks may be best served by different visualization techniques.

- A final use for studies in visualization is to show that an abstract theory applies under certain practical conditions.
  For example, results from psychophysics or computer vision may or may not extend to a visualization environment. User studies can be run to test this hypothesis. Results can show *when* the theories hold, and *how* they need to be modified to function correctly for real world data and tasks.

3. Limitations of user studies

While user studies are an important tool for visualization design, they are not the proper choice in every situation. In this section, we will see some limitations of user studies.

First of all, the quantitative user studies have a clear role in visualization, they may not always be the right choice. Therefore, it is important to consider other options before jumping in to design and run a user study.

Secondly, studies are very time consuming to design, implement, run, and analyse. Typically, they can only be used to answer small questions, and any larger conclusions rely on generalizations that may not be valid.

Last but not least, for some studies, experimental design may lead to results that are not statistically significant. Some studies are not published because of null results, or because the results are inconclusive or not compelling.

4. Conclusion

To sum up, user studies can improve the quality of research since we normally strive for effective visualizations. Although it is difficult to design a good experiment, and the relevant skills require substantial study tempered with experience, a well conducted study is usually worth the effort. The results can ultimately have a considerable impact and potentially contribute to the scientific foundations of the discipline.

Even though we advocate more user studies, we recognize that other methods are available that may be more appropriate in certain situations. We should be aware of these methods, so we can select the best tool for the problem at hand.

Javascript and HTML5 Canvas 2

The objective of this section is present some key points of Javascript and HTML5 Canvas which are presented in a video lecture of professor Wei Tsang Ooi [4]. This video mentions two main problems, they are: Object Oriented Programming in Javascript and drawing, controlling an object with HTML5 Canvas. I reserve the section 1 to bring up Object Oriented Programming in Javascript and the section 2 to touch on HTML5 Canvas. In the section 3, I implement a version of Pong to practise the knowledge learned from this video.

1. Object Oriented Programming in Javascript

1.1. Javascript class definition

In Javascript, there is no classes and functions can be used to somewhat simulate classes. To define a class in Javascript, we define a normal Javascript function and then create an object by using the `new` keyword, as seen in code 2.1.

```
1 // define class Ball
2 function Ball(){
3     // content of class
4 }
5
6 // create an object of Ball
7 var b = new Ball();
```

Code 2.1: Javascript class definition

1.2. Types of property

There are 3 types of property in Javascript class: `private`, `public`, and `static`. *Private properties* can be accessed only by methods of class. *Public properties* can be accessed by an object of class. In private methods, we cannot call a public property. To get value of a public property we have to use privilege methods. *Static properties* are declared outside the definition of class.

To define private properties of a class, we declare these properties as normal variables in function by using `var` keyword. In order to define public properties of a class, we use

the keyword `this`. Last but not least, to define static properties of a class, we use `dot` operator, as seen in code 2.2.

```
// define class Ball
function Ball(){
    // declare private properties: x, y, vx, vy, canvas_id, that
    var x; var y;
    var vx; var vy;
    var canvas_id;
    var that = this;

    // declare public properties: width, height
    this.width;
    this.height;
}

// declare static properties: VX, VY, WIDTH, HEIGHT
Ball.VX = 1;
Ball.VY = 1;
Ball.WIDTH = 800;
Ball.HEIGHT = 600;
```

Code 2.2: Types of property in Javascript class

1.3. Types of method

In Javascript class, there are 3 types of method: `private`, `privilege`, and `static` method. Private and privilege methods are declared inside the definition of class. *Private methods* are declared by using `var` keyword and *privilege methods* are declared by using `this` keyword.

Static methods are declared outside the definition of class by using `dot` operator. The code 2.3 shows the way to declare these 3 types of method.

```
// define class Ball
function Ball(){
    // declare private methods: move, inc_vy, dec_vy, inc_vx, dec_vx
    var move = function(){
        if((x >= that.width) || (x < 0)) vx = -vx;
        x += vx;

        if((y >= that.height) || (y < 0)) vy = -vy;
        y += vy;
        that.print();
    }

    var inc_vy = function(){vy += 1;}
    var dec_vy = function(){vy -= 1;}
    var inc_vx = function(){vx += 1;}
```

```
16      var dec_vx = function(){vx -= 1;}
17
18      // declare privilege methods: play, set_xy, set_size, set_id
19      this.play = function(){
20          setInterval(function(){move()}, 100);
21      }
22
23      this.set_xy = function(x1, y1){
24          x = x1;
25          y = y1;
26      }
27
28      this.set_id = function(id){
29          canvas_id = id;
30      }
31  }
32
33  // declare static method CONSTRUCTOR
34  Ball.CONSTRUCTOR = function(){
35      // declare static properties: VX, VY, WIDTH, HEIGHT
36      Ball.VX = 1;
37      Ball.VY = 1;
38      Ball.WIDTH = 800;
39      Ball.HEIGHT = 600;
40  }
```

Code 2.3: Types of method in Javascript class

1.4. Experimentation

In this section, I will do some tests with types of property and method to make it more clearly.

1.4.1. Access private property

```
1  "use strict";
2
3  // declare class Ball
4  function Ball(){
5      // declare private properties: priX, priY, vx, vy
6      var priX; var priY;
7      var vx; var vy;
8
9      // constructor
10     priX = 0; priY = 0;
11     vx = 1; vy = 1;
12     setInterval(function(){move();}, 1000);
13
14     // declare private method: move
15     var move = function(){
16         priX += vx;
17         priY += vy;
18         priPrint();
19     }
20
21     var priPrint = function(){
22         console.log("from private method: (priX, priY) = (" + priX + "," + priY + ")");
23     }
24
25     this.pubPrint = function(){
26         console.log("from public method: (priX, priY) = (" + priX + "," + priY + ")");
27     }
28 }
29
30 var b = new Ball();
31 console.log("priX = " + b.priX);
32 b.pubPrint();
```

Figure 2.1: Code source

```
vatuan@vatuan-XPS: /media/DATA/Documents/USTH/Semester2/SE4IM/TP/lab2
vatuan@vatuan-XPS:/media/DATA/Documents/USTH/Semester2/SE4IM/TP/lab2$ node exper.js
priX = undefined
from public method: (priX, priY) = (0,0)
from private method: (priX, priY) = (1,1)
from private method: (priX, priY) = (2,2)
from private method: (priX, priY) = (3,3)
from private method: (priX, priY) = (4,4)
from private method: (priX, priY) = (5,5)
from private method: (priX, priY) = (6,6)
from private method: (priX, priY) = (7,7)
from private method: (priX, priY) = (8,8)
from private method: (priX, priY) = (9,9)
from private method: (priX, priY) = (10,10)
from private method: (priX, priY) = (11,11)
from private method: (priX, priY) = (12,12)
from private method: (priX, priY) = (13,13)
from private method: (priX, priY) = (14,14)
from private method: (priX, priY) = (15,15)
from private method: (priX, priY) = (16,16)
from private method: (priX, priY) = (17,17)
from private method: (priX, priY) = (18,18)
from private method: (priX, priY) = (19,19)
from private method: (priX, priY) = (20,20)
^Cvatuan@vatuan-XPS:/media/DATA/Documents/USTH/Semester2/SE4IM/TP/lab2$
```

Figure 2.2: Result

In the code presented in figure 2.1 I create 4 private properties of class `Ball` and use constructor to initialize its value. Function `move()` is used to change values of properties, and 2 functions `priPrint()` and `pubPrint()` are used to show them in console. In line 30, I create an instance of class `Ball` and try to access private property `priX` in line 31. I call public method `pubPrint()` in line 32.

The result of this code is shown in figure 2.2. The line `priX = undefined` (the 2^{nd} line) in the result demonstrates that we cannot access a private property from an instance of class. The line `from public method: (priX, priY) = (0,0)` (the 3^{rd} line) and line `from private method: (priX, priY) = (1,1)` (the 4^{th} line) evince that we can access a private property from private and public method of class.

1.4.2. Call public method

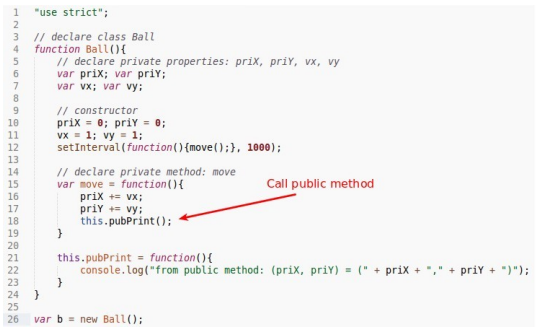


Figure 2.3: Code source

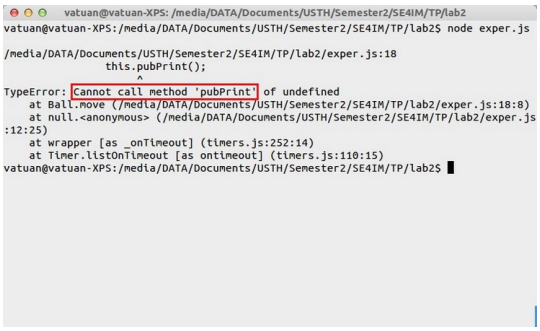


Figure 2.4: Result

In the code presented in figure 2.3 I try to a public method (`pubPrint()` method) from a private method (`move()` method). The result in figure 2.4 shows that we cannot call directly a public method from a private method. To pass this problem, we can change the private method (`move()` method) to public method. We still have another way that is presented in the figure 2.5.

Figure 2.5: Success code

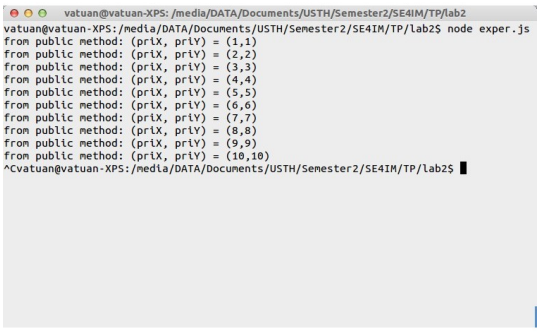


Figure 2.6: Result

In this way, I create a private variable `that` and assign its value to `this` pointer. Then, in private method `move()` I call public method `pubPrint()` over this private variable.

2. HTML5 Canvas

The HTML5 Canvas element `<canvas>` is used to draw graphics, on the fly, via scripting (usually Javascript). The canvas element is only a container for graphics. We must use a script to draw and control the graphics.[5]

A canvas is a rectangular area on an HTML page, and it is specified with the `<canvas>` element. By default, the <canvas> element has no border and no content.

```
<canvas id="myCanvas" width="800" height="600"></canvas>
```

Code 2.4: Create a Canvas

The code 2.4 shows the way to create a canvas that is a rectangle 800 x 600 pixels. In this code, the attributes `width` and `height` tell browser the size of canvas. The attributes `id` tell browser the name of canvas. This attribute is used to distinguish one from other when we have more than one canvas in a page.

2.1. Drawing graphics

As presented above, the canvas element is only a container for graphics. Therefore, to draw graphics we have to use a script. The code 2.5 illustrates how to draw graphics in canvas. In this case, the code set two attributes: width and height for canvas and draw a green circle.

```
1  this.print = function(){
2          var playArea = document.getElementById(canvas_id);
3          playArea.width = this.width;
4          playArea.height = this.height;
5
6          var context = playArea.getContext("2d");
7          context.fillStyle = 'green';
8          context.beginPath();
9          context.arc(x, y, 10, 0, Math.PI*2, true);
10         context.closePath();
11         context.fill();
12     }
```

Code 2.5: Draw graphics

In paragraph bellow, I explain more detailed the code 2.5. The result of this code is present in figure 2.7.

- Line 2 create a variable which represent for an element that has `id` is value of variable `canvas_id`. In this case, it is id of a canvas.

- Lines 3, 4 set the size of canvas.
- Line 6 get object `getContext("2d")` - this is a built-in HTML5 object to variable `context`.
- Line 7 set the property `fillStyle` of `context` to green color.
- Lines 8, 9, 10 use method `arc()` to draw a circle on canvas `playArea`. This method takes 5 parameters: `x, y` is coordinates of the centre of circle; `10` is the radius of circle; `0, Math.PI*2` are corresponding start and end angle of arc; and `true` is to said that draw the arc anticlockwise.
- Line 11 is to draw a solid circle with color is set in `fillStyle`.

2.2. Controlling graphics

We have a circle that is drew by method `print()` in code 2.5. We also have two methods:

- method `move()` which is presented at line 4 in code 2.3 is to displace the circle from position `(x,y)` with the velocity of corresponding coordinates are `vx, vy`.
- method `play()` which is presented at line 19 in code 2.3 is to call method `move()` every 100 miliseconds.

The code 2.6 show the way that we used to control the movement of this circle. In details, we can change the velocity of the movement of the circle by using the keyboard. To do this, we listen event `keydown` on the document (the web page). Based on the `keyCode` we know what key is pressed, so we can increase or decrease the velocity of `x` or `y` corresponding with the key pressed.

```
1  // control movement
2    document.addEventListener("keydown", function(e){
3        switch (e.keyCode){
4            case 37: // left arrow
5                dec_vx();
6                break;
7            case 38: // up arrow
8                dec_vy();
9                break;
10           case 39: // right arrow
11               inc_vx();
12               break;
13           case 40: // down arrow
14               inc_vy();
15               break;
16       }
17   }, false);
```

Code 2.6: Control movement

3. Implementation

Following the video tutorial, I implement a version of Pong to practise the learned knowledge about Object Oriented Programming in Javascript and drawing, controlling a graphics in HTML5 Canvas.

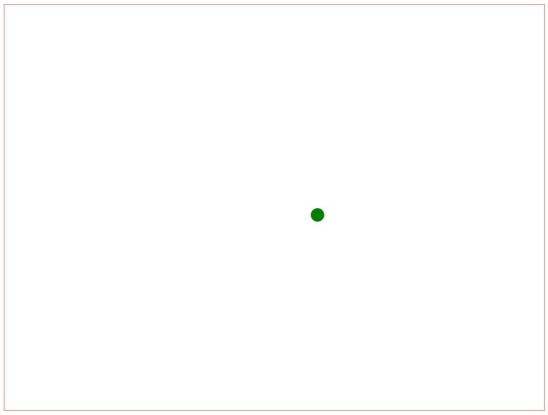


Figure 2.7: Pong implementation

In my own version of Pong, there are some improvements in comparison with the tutorial:

- the ball start at a random position in canvas instead of always starting at upper left hand corner.
- the ball automatic change the direction when its center touch on the border of canvas. It cannot move outside the canvas.
- the motion of the ball is controlled under 4 directions (left, right, up, and down) instead of under 2 directions (up and down) as in the tutorial.
- the script `Ball.js` is improved to used as a library, not only for a fixed id of canvas.

Zooming Video

3

The objective of this section is to understand the function context in Javascript and to practise zooming video in HTML5 with mouse and keyboard. In the section 1 I present my knowledge about function context in Javascript. The section 2 is reserved to touch on two states machine of zooming video with mouse. In the section 3, I go one step further by combining between mouse and keyboard to zoom video in HTML5.

1. Function context

The code 3.1 below gives an example of function context in Javascript. By analysing this code line by line, I will present my knowledge about this problem.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>VU Anh Tuan - Function Context Example</title>
5      <script type="text/javascript">
6        var o1 = {handle:'o1'};
7        var o2 = {handle:'o2'};
8        var o3 = {handle:'o3'};
9        window.handle = 'window';
10
11       function whoAmI(){ return this.handle; }
12
13       o1.identifyMe = whoAmI;
14
15       alert(whoAmI());
16       alert(o1.identifyMe());
17       alert(whoAmI.call(o2));
18       alert(whoAmI.apply(o3));
19     </script>
20   </head>
21   <body>
22   </body>
23 </html>
```

Code 3.1: Function Context Example

In three lines from 6 to 8, they define three objects named `o1, o2`, and `o3`. Each object has a property named `handle` and this property of these objects have corresponding

values *o1, o2*, and *o3*.

```
6  var o1 = {handle:'o1'};
7  var o2 = {handle:'o2'};
8  var o3 = {handle:'o3'};
```

In line 9, they create property `handle` of predefined object `window` and put value *window* on it.

```
9  window.handle = 'window';
```

In line 11, they define function `whoAmI()` in order to return the value of property `handle` of object who calls it.

```
11 function whoAmI(){ return this.handle; }
```

In line 13, they create property `identifyMe` of object `o1` and assign its value to the instance of function `whoAmI()`.

```
13 o1.identifyMe = whoAmI;
```

In line 15, they alert to content of function `whoAmI()`. Since in this line, they do not indicate the object who calls function `whoAmI()` and this function is called in top-level scope, browser understands that function `whoAmI()` is called by the predefined `window` object. Thus, return value of function `whoAmI()` in this case is *window* (it is defined in line 9 of code 3.1).

```
15 alert(whoAmI());
```

Figure 3.1: Result of command `alert(whoAmI())`

Since the value of property `identifyMe` of object `o1` is the instance of function `whoAmI()`, line 16 will alert the value of property `handle` of object `o1` because `o1` is object who call function `whoAmI()`. Therefore, in the result it will alert *o1* (defined in line 6 of code 3.1).

```
16  alert(o1.identifyMe());
```

Figure 3.2: Result of command `alert(o1.identifyMe())`

Since in Javascript, everything is object, a function hence has its own properties and methods. `call()` and `apply()` are two of predefined methods in every object of Javascript.

In line 17, the function `whoAmI()` invokes the predefined method `call()` and uses its first parameter *o2* as the *this* pointer inside the body of the function (line 11 of code 3.1). In other words, the command `whoAmI.call(o2)` has told the runtime that the object `o2` will be referenced as *this* while executing inside of function `whoAmI()`. Thus, in the result this line will alert the value of property `handle` of object `o2`. It is *o2* (defined in line 7 of code 3.1).

```
17  alert(whoAmI.call(o2));
```

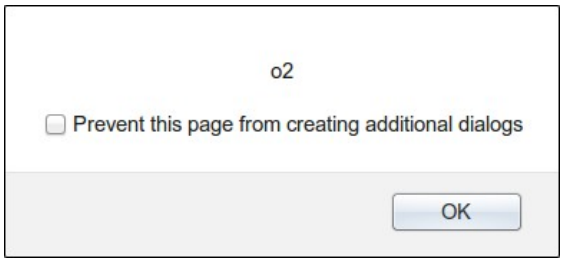


Figure 3.3: Result of command `alert(whoAmI.call(o2))`

The predefined method `apply()` is identical to method `call()`, except this method requires an array as the second parameter. The second parameter of the method `apply()`

is used if function who call this method need at least one argument. In our case, the function `whoAmI()` do not need any argument, so we saw that the use of method `apply()` is identical to the one of `call()` method.

In line 18, the function `whoAmI()` invokes the predefined method `apply()` to announce that it uses the object *o3* as the *this* pointer inside the body of the function (line 11 of code 3.1). Thus, in the result this line will alert the value of property `handle` of object `o3`. It is *o3* (defined in line 8 of code 3.1).

```
18  alert(whoAmI.apply(o3));
```

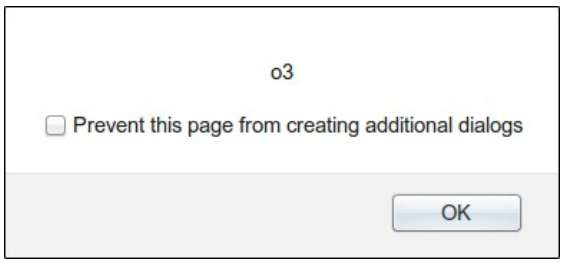


Figure 3.4: Result of command `alert(whoAmI.apply(o3))`

To sum up, the code 3.1 will alert respectively 4 message: *window, o1, o2, o3* to the screen.

If we add a 5^{th} alert in 3.2 to line 19 of code 3.1 it will alert one more message *o3*.

```
18  alert(o1.identifyMe.call(o3));
```

Code 3.2: $5^{th} alert$

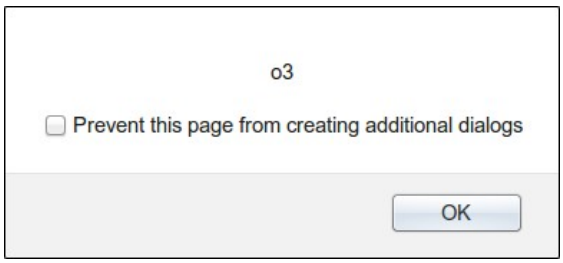


Figure 3.5: Result of command `alert(o1.identifyMe.call(o3))`

This is easy understand because as I explained before, the value of `o1.identifyMe` is a function instance of function `whoAmI()`, so the $5^{th} alert$ is equivalent to this command: `alert(whoAmI.call(o3));` Thus, it will alert to the screen the value of `o3.handle`. It is *o3*.

2. Two states mechanism

2.1. Problem

In this section I implement the two states mechanism (*zoomed* and *not zoomed*) for a zoomable video. The middle mouse button is used to change the state. If current state is *zoomed*, a middle mouse click on the video will change it to the *not zoomed* state. Conversely, when the video is playing in state *not zoomed* state, a middle mouse click on the video will change it to the *zoomed* state. I use the middle button instead of left or right button in order to avoid the conflict with controlling (play and pause) video of left button and with showing context menu of right button.

By default, the video plays in *not zoomed* state (figure 3.6). In this state, whole video (figure 3.6a) is presented in the canvas with size 640x360 pixels (figure 3.6b).

(a) The video (b) The canvas

Figure 3.6: In *not zoomed* state

In *zoomed* state (figure 3.7), a region of interest (ROI) inside the video (rounded rectangle in figure 3.7a) with size 320x180 pixels will zoomed to 200% in the canvas (figure 3.7b). The ROI takes position of mouse when middle button clicked in the video as center of region.

(a) The video (b) The canvas

Figure 3.7: In *zoomed* state

2.2. Implementation

2.2.1. HTML

To implement this two states mechanism, I create 3 objects in HTML: `video, canvas`, and `roi`. The object `video` is to control the video (play, pause ...), the object `canvas` is to display the zoomed region inside the video in *zoomed* state and display the entire video in *not zoomed* state. The object `roi` is to show the region inside the video which is zoomed in *zoomed* state. It helps user to facilitate determining the region of interest inside the video.

```
<!DOCTYPE html>
<html>
<head>
    <title>VU Anh Tuan - Two States Mechanism</title>
    <script type="text/javascript" src="twostates.js"></script>
    <style>
        #roi{
            width: 320px;
            height: 180px;
            position: absolute;
            overflow: hidden;
            border-radius: 10px;
            display: none;
            background: #FFF;
            opacity: 0.3;
        }
    </style>
</head>
<body onload="initialize();">
    <video id="video" src="ct640.ogg" width="640" height="360" controls
    autobuffer></video>
    <canvas id="canvas" width="640" height="360"></canvas>
    <div id="roi"></div>
</body>
</html>
```

Code 3.3: HTML for two states mechanism

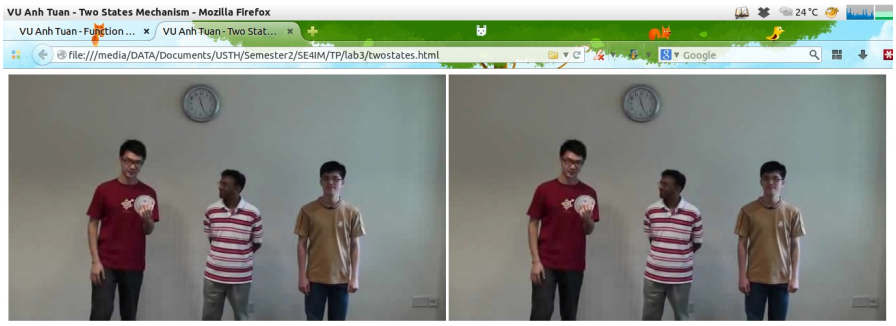


Figure 3.8: *not zoomed* state

Figure 3.9: *zoomed* state

2.2.2. Javascript

When the HTML file loads the function `initialize()` is called. At the beginning, this function assigns default value to global variables: `video`, `videoPos`, `canvas`, `context`, `roi` and `zoomed`.

```
1  function initialize(){
2  //     Variables initialization
3      video = document.getElementById("video");
4      videoPos = findPosition(video);
5      canvas = document.getElementById("canvas");
6      context = canvas.getContext("2d");
7      roi = document.getElementById("roi");
8      roi.width = 320;
9      roi.height = 180;
10     zoomed = false;
11     video.addEventListener("play", timerCallback, false);
12     video.addEventListener("mousedown", function(event){
13         if(event.button == 1){ handleClick(event); }
14     }, false);
15     roi.addEventListener("mousedown", function(event){
16         if(event.button == 0){ videoControl(); }
17         if(event.button == 1){ handleClick(event); }
18     }, false);
19 }
```

Code 3.4: function `initialize()`

After that, the function `timerCallback()` is attached to event *play* of video in order to synchronize instantly the image in canvas with frames of video. This function call function `computeFrame()` to determine whole frame of video or only ROI of frame is captured to the canvas. If the video is playing in the state *not zoomed*, whole frame will be shown (line 2-4 of the code 3.6). Contrariwise, the ROI of frame will be zoomed to 200% (line 5-7 of the code 3.6).

```
function timerCallback(){
    computeFrame();
    setTimeout(function(){
        timerCallback();
    }, 0);
}
```

Code 3.5: function `timerCallback()`

```
function computeFrame(){
    if(!zoomed){
        context.drawImage(video, 0, 0, video.width, video.height);
    }
    else{
        context.drawImage(video, roi.offsetLeft - videoPos[0], roi.
    offsetTop - videoPos[1], roi.width, roi.height, 0, 0, canvas.width,
    canvas.height);
    }
}
```

Code 3.6: function `computeFrame()`

(a) whole frame (b) ROI of frame

Figure 3.10: captured frame to the canvas

In function `initialize()`, function `handleClick()` is attached to event middle mouse button clicked inside the video (line 12-14, function `initialize()`) in order to change state from *zoomed* to *not zoomed* each time middle mouse button is clicked on the video and vice versa.

```
function handleClick(event){
    if(zoomed){
        zoomOut(event);
    } else{
        zoomIn(event);
    }
}
```

Code 3.7: function `handleClick()`

(a) zoom in (b) zoom out

Figure 3.11: zooming video

In the *zoomed* state, when the middle mouse button clicks on the video the function `handleClick()` calls function `zoomOut()` to change current state to the *not zoomed* state and hide the ROI inside the video.

```
function zoomOut(event){
    zoomed = false;
    roi.style.display = "none"; // hide ROI
}
```

Code 3.8: function `zoomOut()`

In the *not zoomed* state, when the middle mouse button clicks on the video the function `handleClick()` calls function function `zoomIn()` to change current state to the *zoomed* state and to show the ROI inside the video by using function `drawROI()`.

```
function zoomIn(event){
    var pagePosition = new Array(event.clientX, event.clientY); //
    position of the mouse in the page
    var mousePosition = getMousePosition(pagePosition, video); //
    position of mouse inside video
    zoomed = true;
    drawROI(mousePosition[0] - roi.width/2 + videoPos[0], mousePosition
    [1] - roi.height/2 + videoPos[1]);
}
```

Code 3.9: function `zoomIn()`

The function function `drawROI()` takes position of mouse as centre of ROI. Since position of mouse is centre of the ROI, when a user clicks on a side of the video, the ROI will overflow outside of the video. To prevent this, I define two variable: `maxLeft, maxTop` in this function to limit ordinates of the ROI.

```
function drawROI(left, top){
    var maxLeft = video.width - roi.width + videoPos[0];
```

```
3       var maxTop = video.height - roi.height + videoPos[1];
4   //    prevent ROI displaying outside the video
5       if(left < videoPos[0]) left = videoPos[0];
6       if(left > maxLeft) left = maxLeft;
7       if(top < videoPos[1]) top = videoPos[1];
8       if(top > maxTop) top = maxTop;
9   //    display ROI
10      roi.style.left = left + "px";
11      roi.style.top = top + "px";
12      roi.style.display = "block";
13  }
```

Code 3.10: function `drawROI()`

Since ROI overlays the video, I attach function `videoControl()` to event left click on ROI (line 16 in function `initialize()`) in order to help user play and pause video as clicking on the video. I also attach function function `handleClick()` to event middle click on ROI (line 17 in function `initialize()`) to help user change state as clicking on the video.

```
1   function videoControl(){
2       if(video.paused){ video.play(); }
3       else{ video.pause(); }
4   }
```

Code 3.11: function `videoControl()`

3. One step further

3.1. Problem

In this section I improve a little the two states mechanism (section 2) in order to use both keyboard and mouse to control the ROI and to use keyboard to vary the scale of the ROI instead of fixing it at 200% like in the section 2. Like in the section 2, user also uses the middle mouse button to active/inactive *zoomed* state. Furthermore, user can use arrow keys to move the ROI and use `+/-` keys to zoom out and zoom in the ROI.

By default, when user activates the *zoomed* state, the ROI is presented in the canvas at scale 1.0. User uses `+/-` keys to zoom in and zoom out the ROI with the scale varied from 0.5 to 2.5. Each time user presses on the button `+/-` on the keyboard, the scale is increased/decreased by 0.1. User also uses arrow keys to displace the ROI in the video. Each time user presses on a arrow key, the ROI move 1 pixel in the direction of that arrow key. In order to avoid the conflict with `prev` and `next` when user use arrow keys in the video, I also set up keys: `A, W, S` and `D` to move the ROI as showing in the figure 3.12. The arrow keys, `+/-, A, W, S` and `D` work only in *zoomed* state. In the *not zoomed* state, none of these keys works.

Figure 3.12: Active keys to control zooming

3.2. Implementation

3.2.1. HTML

Like in the section 2.2.1, I also create 3 objects in HTML: `video`, `canvas` and `roi`: one for controlling the video, one for presenting the ROI and one for showing the ROI overlays the video. The only difference between HTML code of this section and the one of the section 2.2.1 is that I create an object called `container` (line 27 in code 3.12) to show current scale of the ROI in the canvas.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>VU Anh Tuan - Step Further</title>
5      <script type="text/javascript" src="stepfurther.js"></script>
6      <style>
7          #roi{
8              width: 256px;
9              height: 144px;
10             position: absolute;
11             overflow: hidden;
12             border-radius: 10px;
13             display: none;
14             background: #FFF;
15             opacity: 0.3;
16         }
17         #container{
18             margin-left: 935px;
19             display: none;
20         }
21     </style>
```

```
</head>
<body onload="initialize();">
    <video id="video" src="ct640.ogg" width="640" height="360" controls
    autobuffer></video>
    <div style="display: inline;">
        <canvas id="canvas" width="640" height="360"></canvas>
        <div id="container">scale: <span id="ratio">1</span></div>
    </div>

    <div id="roi"></div>
</body>
</html>
```

Code 3.12: HTML for step further

3.2.2. Javascript

There is a little difference between Javascirpts of this implementation and the one in the section 2.2.2. The big differences are in the function `computeFrame()` and in the added event keydown on the document.

```
function computeFrame(){
    if(!zoomed){
        context.drawImage(video, 0, 0, video.width, video.height);
    }
    else{
        if(scale < 0.5) scale = 0.5;
        if(scale > 2.5) scale = 2.5;
        var draw = new Object();
        draw.width = roi.width * scale;
        draw.height = roi.height * scale;
        draw.left = (canvas.width - draw.width)/2;
        draw.top = (canvas.height - draw.height)/2;
        context.drawImage(video, roi.offsetLeft - videoPos[0], roi.
    offsetTop - videoPos[1], roi.width, roi.height, draw.left, draw.top,
    draw.width, draw.height);
    }
    ratio.innerHTML = scale.toFixed(1);
}
```

Code 3.13: function `computeFrame()`

In the *zoomed* state, each time the function `computeFrame()` called it compute the scale of the ROI in the video before drawing the ROI with scale in the canvas. To limit the scale of the ROI between 0.5 and 2.5, the function automatically set the value of the global variable `scale` to 0.5 if it find this value lower 0.5 and set the this value to 2.5 if it find this value higher 2.5.

In the *not zoomed* state, the function `computeFrame()` does not compute anything. It only shows entire frame of the video in the canvas.

scale: 0.5 scale: 1.0

scale: 2.5

Figure 3.13: The varied scale of a ROI

```
document.addEventListener("keydown", function(e){
    switch (e.keyCode){
        case 37: // left arrow
            if(zoomed){ drawROI(roi.offsetLeft - 1, roi.offsetTop); }
            break;
        case 38: // up arrow
            if(zoomed){ drawROI(roi.offsetLeft, roi.offsetTop - 1); }
            break;
        case 39: // right arrow
            if(zoomed){ drawROI(roi.offsetLeft + 1, roi.offsetTop); }
            break;
        case 40: // down arrow
            if(zoomed){ drawROI(roi.offsetLeft, roi.offsetTop + 1); }
            break;
        case 61: // +
            if(zoomed){
                if(scale < 2){
                    context.clearRect(0, 0, canvas.width, canvas.height);
                }
                scale += 0.1;
            }
            break;
        case 173: // -
            if(zoomed){
                if(scale > 0.5){
                    context.clearRect(0, 0, canvas.width, canvas.height);
                }
                scale -= 0.1;
            }
            break;
        case 65: // A
            if(zoomed){ drawROI(roi.offsetLeft - 1, roi.offsetTop); }
            break;
        case 87: // W
            if(zoomed){ drawROI(roi.offsetLeft, roi.offsetTop - 1); }
            break;
        case 68: // D
            if(zoomed){ drawROI(roi.offsetLeft + 1, roi.offsetTop); }
            break;
        case 83: // S
            if(zoomed){ drawROI(roi.offsetLeft, roi.offsetTop + 1); }
            break;
    }
}, false);
```

Code 3.14: event keydown

In the event keydown on the document, I assign event of arrow keys and `A, W, D, S` keys pressed to move the ROI and assign event of `+/-` keys pressed to increase/decrease scale of the ROI.

SmartPlayer: User-Centric Video Fast-Forwarding

4

The objective of this section is present a summary of article *Smart Player: User-Centric Video Fast-Forwarding* [7]. The rest of this section is organized as follows. To begin with, I introduce the context and the problem of current video players, then the objective of authors' article in section 1. I reserve section 2 to present some related approaches and related works. Latterly, I enter the authors' solution in section 3. In this section, I also mention their user studies tasks to design and evaluate their SmartPlayer. Finally, I give related knowledge in this article to the course MM2.1 *Software Engineering for Interactive Media* in section 4.

1. Introduction

These days, the developments of the Internet in particular and of digital technologies in general have made it easy for people to download, record and watch videos on a variety of media access devices. Some smart media access devices such as the TiVo set-up box (figure 4.1) even learns users' preference about TV programs, and automatically record them for users [8]. In addition, mass storage devices are inexpensive, so users are easy to collect a lot of videos that they like on a storage device (figure 4.2) such as hard disk, usb, memory card, etc.

Figure 4.1: TiVo set-up box

Figure 4.2: Storage devices

Although digital content recording and storing technologies continue to improve over

time, video playback systems have not changed much. Commercial video players such as QuickTime Player, CyberLink PowerDVD, Microsoft Windows Media Player... offer comparable sets of simple controls for playing, pausing, stopping, fast-forwarding, and rewinding/reversing videos.

When users have limited patience or time to watch the entire length of a video, they are obliged to manually skim and fast-forward to locate content of interest to watch in fine detail. Hence, there is a great need for a new smart playback mechanism which helps users efficiently skim through and fast-forward lengthy and boring content while slowing down to watch the good parts in fine detail.

In the article [7], authors proposed a new video interaction model to solve this problem. It called *adaptive fast-forwarding* model. This model helps people quickly browse videos with predefined semantic rules. The basic idea of the model rises from the metaphor of *scenic car driving* in which drivers adjust their speed according to road conditions as well as the quality of the scenery. When the scenery is monotonous or boring, drivers tend to speed up and skip it. When the scenery is complex and interesting, drivers tend to slows down to get a better look.

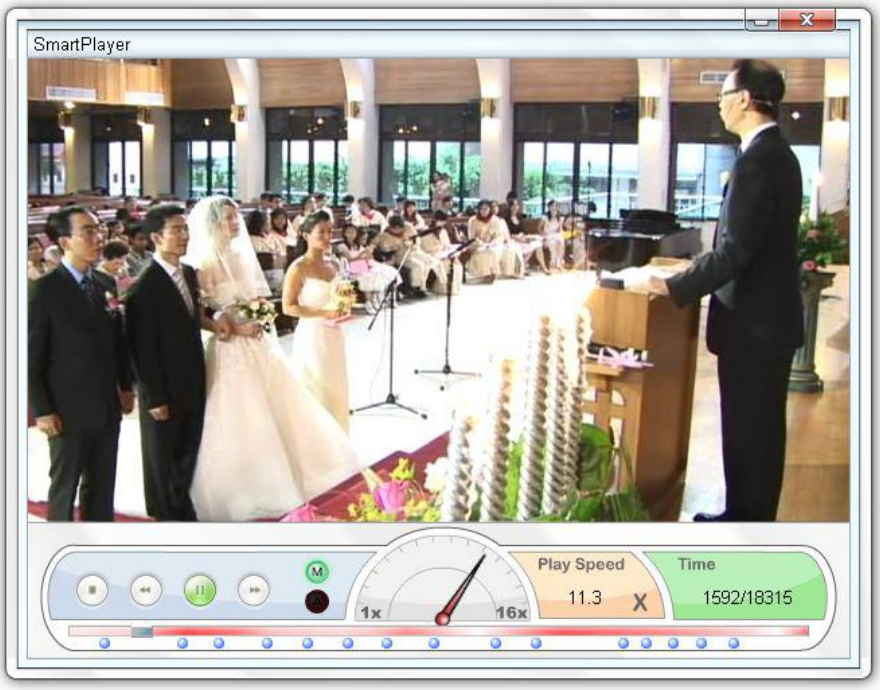


Figure 4.3: SmartPlayer

In order to evaluate their video interaction model, the authors design a video player called *SmartPlayer* (figure 4.3) and use user studies (section 1) to get responses of end user. Their video player has 3 main features:

- The video player adaptively adjusts playback speed based on the complexity of the current scene and predefined semantic events.
- The video player learns the user's preferences about the predefined semantic event types as well as user's favourite playback speed when watching videos matching these event types. Learning user preferences is necessary for adapting video playback speed.
- The video player plays the video continuously with a user acceptable playback rate so as not to miss any undefined events or areas of interest.

2. State of the art

There are several video summarization methods which enables users to skim through content within a short amount of time. They can be categorized into two approaches: *still-image abstraction* and *video skimming*.

2.1. Still-image abstraction

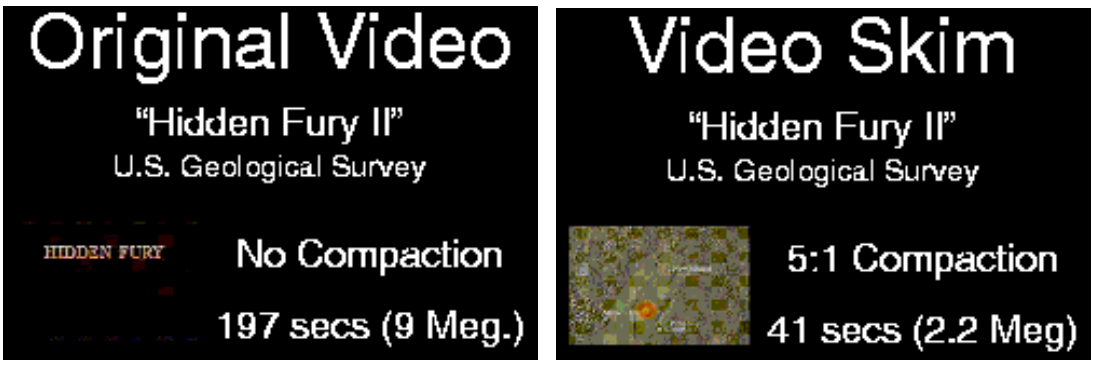
In this approach, key-frames are extracted from a video and used to compose a brief content summary. For example, key-frames can be played back sequentially as a slide show or composed into an image mosaic (figure 4.4). *Still-image abstraction* enable users to quickly obtain a high-level understanding of what is contained in a video.

Figure 4.4: Video on the left. On the right are images mosaic of the video. (*Image:* taptaptap.com)

Liu *et al.* [9] apply video abstraction techniques to identify important frames into a *video collage*. One representation of a video collage is a mosaic, which also serves as a catalog with entry points to different video segments. However, this method does not provide sufficient information to users who want finer details on the parts they are interested in, either for comprehension or entertainment.

2.2. Video skimming

The *video skimming* approach uses automated video analysis to extract segments (figure 4.6) that carry significant information, then compose them into a short video summary (figure 4.5). There are some predefined events or rules which guide the decision on the significance of different video segments.



(a) Origin (b) Compaction

Figure 4.5: The video Hidden Fury II. (*Image:* cs.cmu.edu)

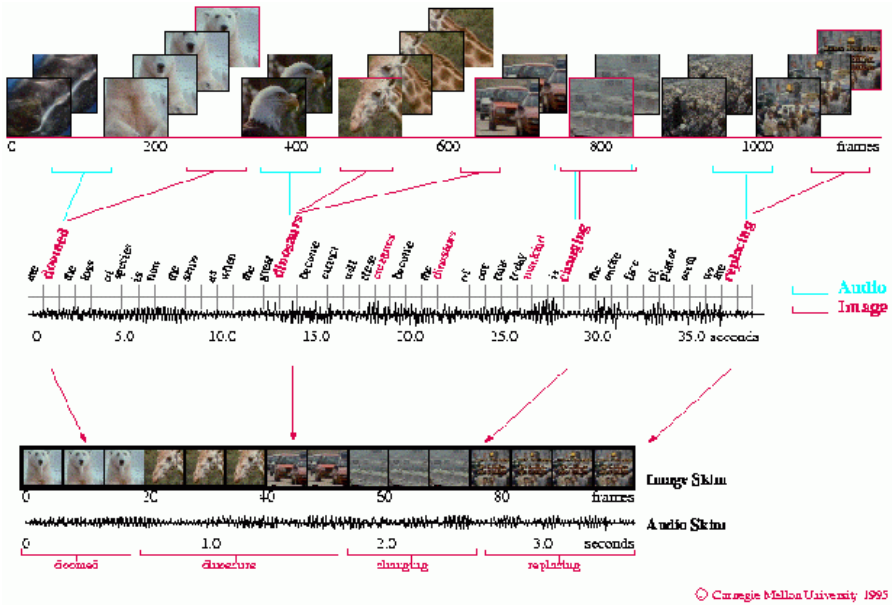


Figure 4.6: Video skimming creation [10]. (*Image:* cs.cmu.edu)

Interesting events of a video are identified by analyzing a variety of image features, including color, contrast, speech, closed captions, camera motion, and human faces. Domain specific knowleadge is often necessary to improve the detection accuracy of

interesting video segments. For example, videos of baseball, tennis, weddings, movies, and news call for analysis of different feature sets corresponding to what are considered interesting semantic concepts in the different video domains.

3. Proposed solution

The authors of the article [7] used the process in the figure 4.7 to design their SmartPlayer. Firstly, they observed user behavior by performing a preliminary user inquiry on how users watch videos with the fast-forwarding mechanism. Then, the authors designed SmartPlayer based on the obtained results. Lastly they recruited test subjects and asked them to perform tasks in order to evaluate how well of SmartPlayer.

Figure 4.7: SmartPlayer Design Processing

3.1. User behavior observation

To observe user behavior the authors recruited 10 unpaid computer-savvy participants (5 males and 5 females) with experience watching videos on computers. Then they performed a user inquiry with them. They concentrated on 2 main questions *what types of video programs that users prefer to fast-forward?* and *how users fast-forward a video?*

Q1. *Why and what types of video programs that users prefer to fast-forward?*

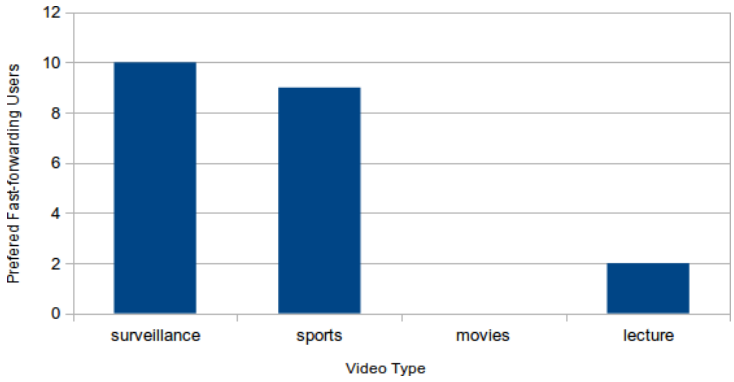


Figure 4.8: Video types prefered to fast-forward

The result of this question is shown in the figure 4.8. All participants fast-forward surveillance videos because these videos are very boring. With relatively simple events videos like sports videos, 9 out of the 10 participants fast-forward because sports videos have predefined patterns and rules which they can use to predict what they are interested in. Only 2 out of the 10 participants fast-forward lecture videos when slides are

shown in the video. The others consider audio critical for understanding lecture videos, and speech comprehension requires time to think and reflect. Therefore, they think that lecture videos should not be fast-forward. At last, no one fast-forwards movies because they expect good movies to be enjoyable from start to finish.

Q2. *How users fast-forward a video?*

To answer this question, the authors designed a prototype player to help user accelerate and decelerate playback speed and to record user playback behavior. Their analysis of the results shows that participants had varying tolerance on the fast-forward speeds for different video types. The user-acceptable fast-forward speed for complex, motion-rich videos (*i.e.,* baseball) was much lower than that of slow videos (*i.e.,* golf). This is because in golf, progress is relatively simple and motion-less, the scene is simple and the camera view does not move or pan too much.

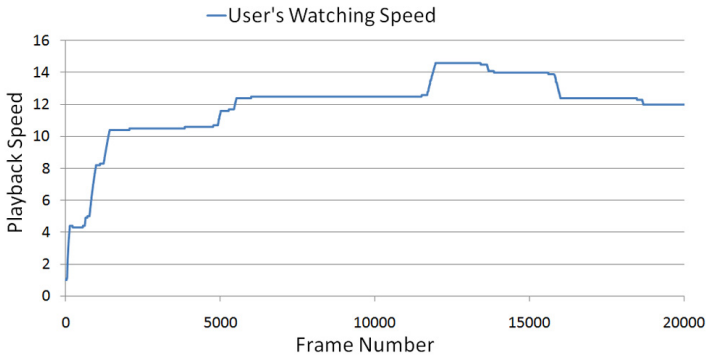


Figure 4.9: One user's watching pattern for a baseball video

Participants in general maintained a constant playback within one video shot (figure 4.9) and seldom changed the playback speed dramatically. They preferred to gradually increase the playback speed, allowing their eyes to accommodate to a higher playback speed. They also prefer not to skip any parts on the first pass, because these in-between video segments provide *context to help them understand what's going on* and enable them to *watch the video more enjoyably.*

To sum up, the results of the observations of user behavior are valuable for designing the SmartPlayer and are summarized below.

- Users tend to maintain a constant playback speed within a video shot when they still want to know what is happening in the video.
- Users prefer gradual rather than sudden or dramatic increases of playback speed.
- Users set the playback rate based on several minutes of recently viewed shots.
- Users prefer not to skip any parts of the video.

3.2. SmartPlayer designing

Based on the results of observations, SmartPlayer does not adjust the playback speed frame by frame. Instead, it cuts the video into a number of segments and then adjusts the playback speed gradually across segment boundaries. SmartPlayer also allows a seemingly continuous playback speed control at a fine increment of 0.1x up to the maximum speed of 16x rather than providing the control of playback speed with discontinuous choices (*i.e.,* playback speed can only be set to 1x, 2x, 4x...) like most existing video players. There is no frame dropping during fast-forwarding so as to preserve the experience of continuous video watching.

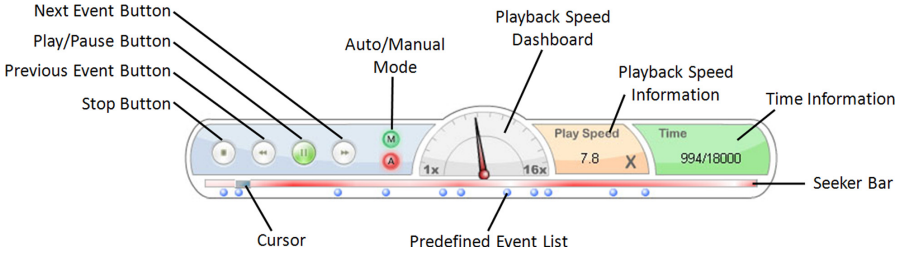


Figure 4.10: The user interface and functions of the SmartPlayer

Figure 4.10 shows the SmartPlayer user interface. In addtion to the basic control buttons (play, pause and stop), the playback speed is shown at the center of the control panel dashboard to match the *scenic car driving* design metaphor. When the playback speed changes, the needle swivels to the current speed. The numeric playback speed is also shown to the right. The seeker bar is used to visualize the scene complexity and semantic events in a video helps users grasp the temporal locations of potential interesting events. The visual scent on the video seeker bar is encoded by the amount of saturation on the red color. If a video segment has a relatively high amount of motion, its red color saturation value on the seeker bar will be higher than those of other video segments. This indicates that the SmartPlayer will likely slow down when playing this motion-rich video segment.

By default, SmartPlayer automatically changes speed according to scene complexity. If users dislike current playback speed under the automatic playing mode, they can manually reset the playback speed. In manual mode, the player adjusts playback speed only according to user input.

To allow for automated playback speed adjustment, the authors developed three software engines: the *motion layer*, the *semantic layer* and the *personalization layer* (figure 4.11).

- The **motion layer** adapts the default playback rate according to detected motion between frames, in which higher motion maps to a lower speed, and vice versa. To support adaptive fast-forwarding, the authors use 2 low-level features for gauging

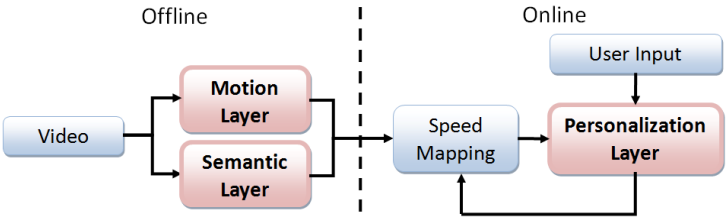


Figure 4.11: Flow of the user centric fast-forwarding mechanism

the similarity between scenes. They are: *color histogram calculating* and *motion estimating*. Calculating color histogram differences between frames allow the authors to detect shot boundaries in a video. To estimate the motion magnitude between two frames, the authors extract optical flows between frames using the Lucas-Kanade method.

- The **semantic layer** detects predefined semantic event points in a video. To effectively extract these event points, predefined domain-specific inference rule are required. As the *semantic layer* is domain-specific, it uses a plug-in framework in which different inference rules can be inserted to process different domain-specific videos. In the testing video clips, the authors used manually annotated semantic events. However, it can be replaced by an automated event detector.

- The **personalization layer** learns user preferences by analyzing the user's previous video browsing behavior. By learning from user input, SmartPlayer updates user preferences with respect to video playback speed. The authors calculate a new video playback speed by linearly interpolating the original playback speed and the user's input speed.
  To facilitate learn user preferences for various event types, at the beginning the SmartPlayer set default playback speeds for all predefined events to the normal speed (1x). If a user dislike one specific event type, he/she will accelerate the playback speed through this specific event. The SmartPlayer thus learns to adjust the playback speed when the same event type is encountered in the future.
  For example, in the figure 4.12 the blue line shows the default playback speed as generated by taking into account each scene's motion complexity and detected predefined events.

3.3. User testing

In this section, the authors tested how well the SmartPlayer improves the user's experience for browsing video by requiring 20 participants (13 males and 7 females) to perform 2 tasks. The first task is for testing the adaptive fast-forwarding mechanism. The second task is for comparing the effectiveness and user satisfactory of the SmartPlayer with that of the traditional player and of the event-based player which plays only system-detected, predefined events and skips other video segments.

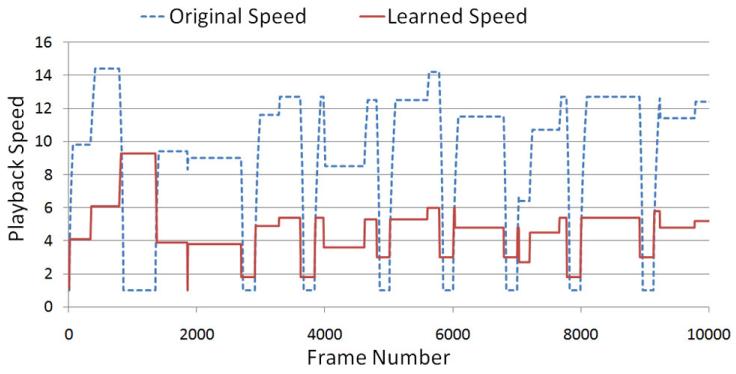


Figure 4.12: Playback speed adjustment

Task 1: Testing the adaptive fast-forwarding mechanism

In this task, participants were asked to watch 25 video clips on 5 types: surveillance, baseball, news, drama and wedding. Each video clip was around 10 minutes long. Participants were required to watch the videos as fast as they could while trying to understand the content.

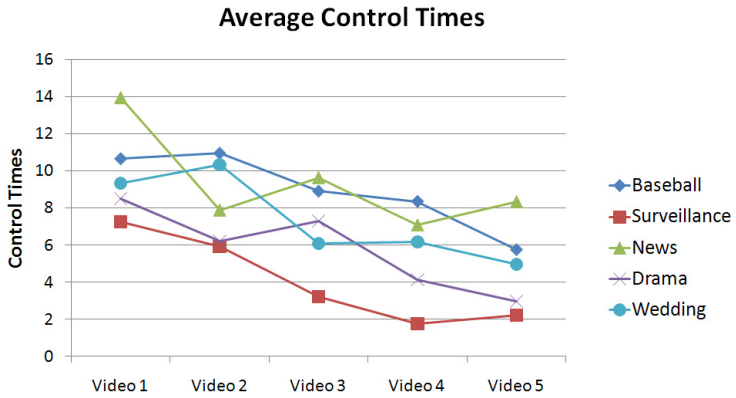


Figure 4.13: Average manual adjustment times

Figure 4.13 shows the average number of manual adjustments for 20 participants to watch 5 video clips in each of 5 video types. From this figure, the authors found that:

- In all 5 video types, the average number of manual adjustments exhibited a decreasing trend from the showing of the first video to the fifth. This suggests that as each participant watched more clips of the same type, the SmartPlayer learned more about his/her preference, thus the resulting in a reduced number of manual adjustments.
- The effectiveness of the SmartPlayer's learning mechanism is different with different video types. With surveillance, baseball and wedding videos the learning

mechanism is better than the others because surveillance videos has explicit events, wedding and baseball videos have explicitly defined rules.

- An unexpected result was that the SmartPlayer's learning mechanism also proved somewhat effective for drama videos. The user interviews show that participants adjusted playback speeds to be no higher than the speed at which they could follow the subtitles in the drama videos. Thus, the bottleneck becomes the playback speed at which viewers can follow the subtitle (approximately 2x to 5x normal playback speed).

Task 2: Comparisons of different video players

In this task, each participant was required to watch 9 video clips on 3 types: baseball, surveillance and news by using 3 video players: SmartPlayer, a traditional player and an event-based player. Each video clips was approximately 10 minutes in length. No one watched the same video clip on different video players. The playback order of video types was set randomly on the different video players.

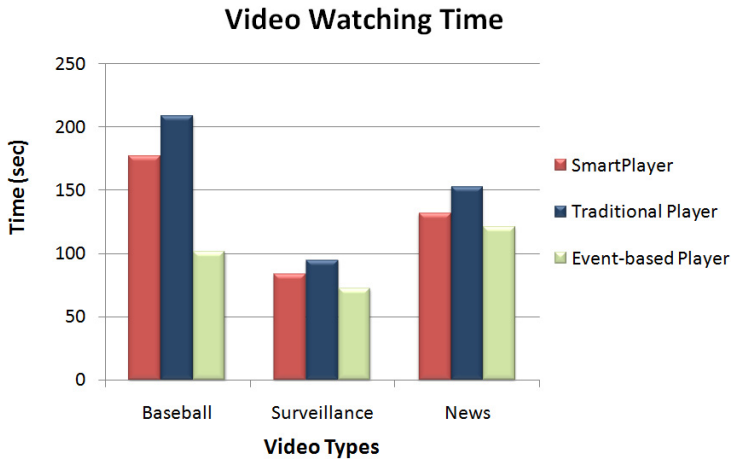


Figure 4.14: Average video watching time

Figure 4.14 shows the average video watching time for the 20 participants. On average, participants spent more watching time using traditional player than that using the SmartPlayer and the event-based player. The event-based player had the the least amount of watching time because it skipped all of the non-event segments and some undefined events. There are 2 main reasons for the traditional player's requiring more time than the SmartPlayer:

- The SmartPlayer provides an event detection mechanism and marks detected events on the seeker bar (figure 4.10). These marks can be seen as good hints to adjust the playback speed while playing videos.
- The SmartPlayer adjusts playback speed according to the scene complexity and detected events.

Hence, when using the traditional player, users do not have any information about what will happen next, and therefore watch the video with relatively slow speeds.

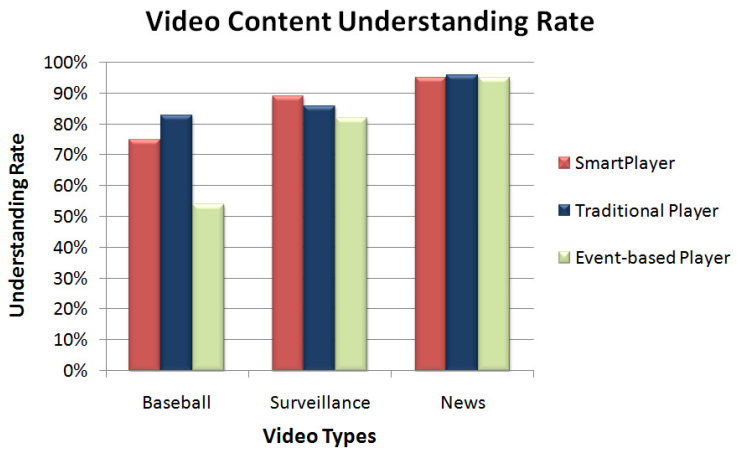


Figure 4.15: Average video content understand rate

Figure 4.15 shows the average video comprehension levels from the 20 participants. From the analysis of this figure, the authors found that:

- On average, participants had better content comprehension using the traditional player than when using the SmartPlayer and the event-based player. The average comprehension level for the SmartPlayer was similar to that of the traditional player. This means that while using the SmartPlayer users can still effectively understand video contents.
- No significant difference was observed for news videos. This is likely because users usually can understand a news story by its title.

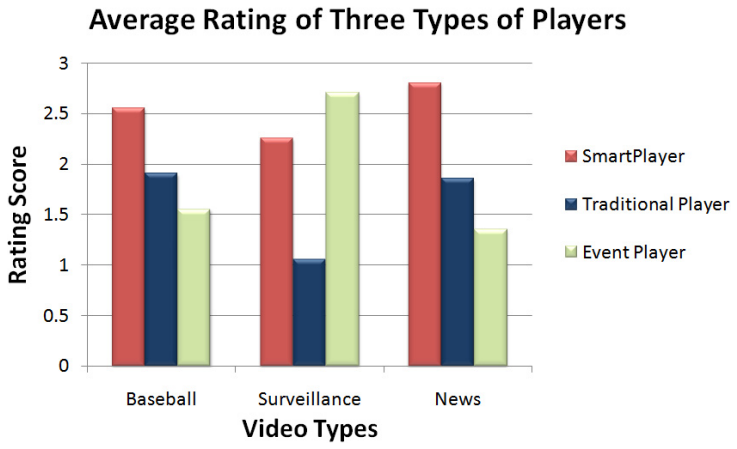


Figure 4.16: Average rating of three types of video players

Figure 4.16 shows the average ratings (a higher score means better preference) calculated from the results of questionnaires filled out by 20 participants for each of the 3 video players in watching 3 video types: baseball, surveillance, and news. For baseball and news videos, participants preferred the SmartPlayer over the others. For surveillance videos, participants preferred the event-based player over the others, because surveillance videos are extremely boring and non-event segments are usually meaningless to viewers.

4. Relation to the course MM2.1

In this article, the authors used 2 knowledge related to the course MM2.1. They are:

- The first is knowledge about *user studies*, which is presented in the section 1 of this report. The authors used user studies to find how users watch videos with the fast-forwarding mechanism and to assess how well the SmartPlayer improves the user's experience to browsing video.
- The second is knowledge about *human-computer interaction*, which is presented in the part 1 of this course. The authors used knowledge of HCI to design SmartPlayer. To begin with, the authors required participants to fill out the questionnaires to find out *what types of video programmes that users prefer to fast-forward* (section 3.1). Then, the authors create a prototype to discover *how users fast-forward a video*. The knowledge of HCI is used to create a questionnaire to rate three types of video players: SmartPlayer, traditional player and event-based player.

5. Conclusion

In this article, the authors proposed an effective playback mechanism called *adaptive fast-forwarding* to skim though and fast-forward lengthy and boring content while slowing down to watch the good parts in fine detail when browsing videos. They also applied this mechanism to design SmartPlayer and compared SmartPlayer with traditional player, event-based player. The results are very good. The SmartPlayer takes less time than traditional player when watching videos while gives the similarly understand the video content. The SmartPlayer is a content-aware player, it give more effectively understand the video content in comparison with event-based player, a non-content-aware player.

This is also a very interesting article. From this article, I learned the process to design a new product from the idea of product to the product testing. This article also introduces a lots of techniques in the image and video processing domain such as color histogram calculating, optical flow estimating, still image abstraction and video skimming...

Visual Time Line

5

1. Advanced Video Player

In this section, I devise an Advanced Video Player (AVP). It contains a built-in video player in HTML5 with a navigation panel (figure 5.1). The navigation panel contains keyframes at every 2 seconds of the video.

My idea to construct this AVP is that: when a video is loaded, a function is called to walk through the video with the step of 2 seconds and capture frames at each step to the navigation panel. Then a function is added to the event left click on the panel to go directly the position of a frame in the time line of the video.

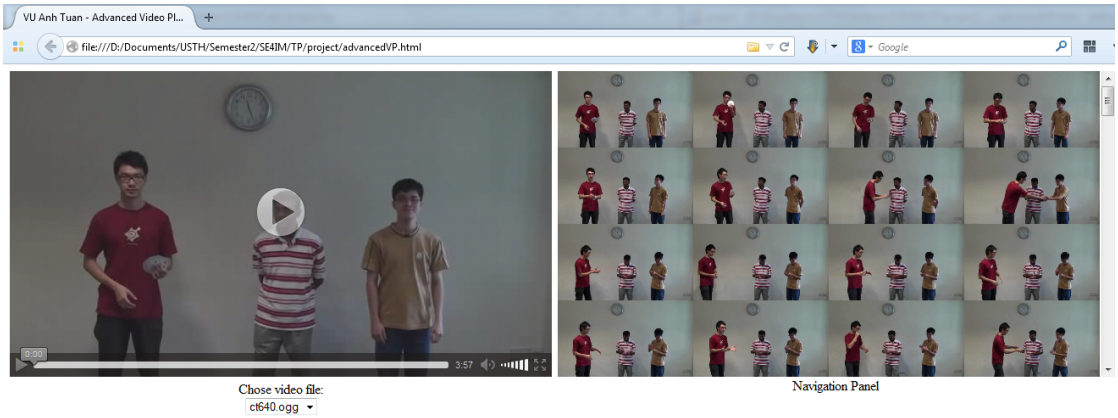


Figure 5.1: Advanced Video Player

1.1. HTML

To implement this advanced video player, I create 3 objects in HTML: `video`, `vifle`, `nav`, and `ptitle`. The object `video` is to control the video (play, pause ...), the object `vfile` is to change video source file. I test my AVP with 3 supplied videos in the course: *bmw.mp4* (figure 5.2a), *ct640.ogg* (figure 5.1) and *ct320.ogg* (figure 5.2b). The object `nav` is to contain keyframes of the video. It is the navigation panel in the AVP. It helps user to facilitate go to a specific frame/moment in the video by clicking on the corresponding image. Lastly, the object `ptitle` is to show the process of constructing

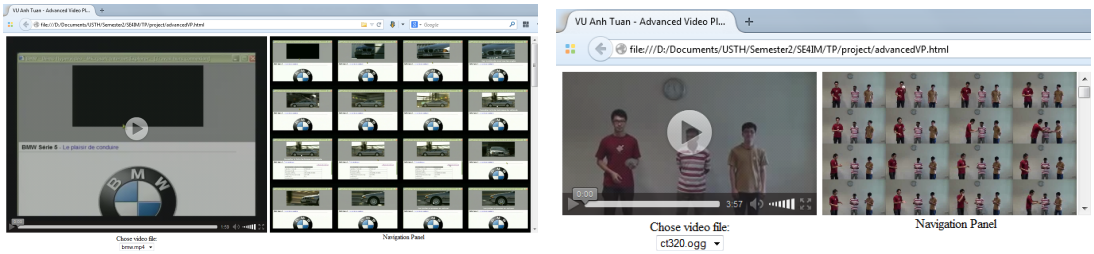
keyframes.

```
<!DOCTYPE html>
<html>
<head>
    <title>VU Anh Tuan - Advanced Video Player</title>
    <script type="text/javascript" src="avp.js"></script>
    <style>
        #container{
            float: left;
            width: 655px;
            height: 360px;
            overflow: auto;
        }
    </style>
</head>
<body>
    <div style="float: left; margin-right: 5px;">
        <video id="video" width="640" height="360" controls autobuffer
    onloadeddata="initialize();">
            <source src="ct640.ogg">
        </video>
        <div style="text-align: center;">
            Chose video file:<br>
            <select id="vfile" autocomplete="off" onchange="changeVideo()
    ;">
                <option value="ct640.ogg">ct640.ogg</option>
                <option value="bmw.mp4">bmw.mp4</option>
                <option value="ct320.ogg">ct320.ogg</option>
            </select>
        </div>
    </div>
    <div style="float: left;text-align: center;">
        <div id="container">
            <canvas id="nav" width="640" height="360"></canvas>
        </div>
        <br>
        <span id="ptitle">Navigation Panel</span>
    </div>
</body>
</html>
```

Code 5.1: HTML for Advanced Video Player

1.2. Javascript

When video file is loaded, function `initialize()` is called. At the beginning, this function assigns values *(line 2 to 9 in the code 5.2)* to global variables: `video, nav, ptitle, context, intVal` and `navCols` (number of columns in the navigation panel). Then it computes values *(line 11 to 14 in code 5.2)* for variables: `fWidth, fHeight` (size of image representative of keyframe in the navigation panel), `numFrames` (number



(a) The video `bmw.png` (b) The video `ct320.ogg`

Figure 5.2: Advanced Video Player with different videos

of keyframes), `navRows` (number of rows in the navigation panel) and assigns values *(line 15 to 16 in code 5.2)* to the size of the navigation panel.

```
1  function initialize(){
2      video = document.getElementById("video");
3      nav = document.getElementById("nav");
4      ptitle = document.getElementById("ptitle");
5      context = nav.getContext("2d");
6      intVal = 2000; // # of milliseconds between frame updates
7      navCols = 4;
8      video.width = video.videoWidth;
9      video.height = video.videoHeight;
10
11     fWidth = Math.floor(video.width/navCols);
12     fHeight = Math.floor(video.height/navCols);
13     numFrames = Math.floor(video.duration*1000/intVal);
14     navRows = Math.ceil(numFrames/navCols);
15     nav.width = video.width;
16     nav.height = navRows*fHeight;
17
18     var container = document.getElementById("container");
19     container.style.width = video.width + 20 + 'px';
20     container.style.height = video.height + 'px';
21
22     construction();
23 }
```

Code 5.2: function `initialize()`

After assigning values to global variables, it calls function `construction()` to build images of keyframes in the navigation panel (figrue 5.3).

There is an interesting thing here. Unlike almost functions of Javascript, the function `drawImage()` is asynchronous. It means that commands which are after the call of function `drawImage()` do not wait the function `drawImage()` complete to run. They run in parallel. Therefore, to construct the navigation panel, we can not make a loop to seek the video to every 2 seconds then capture frame as normal.

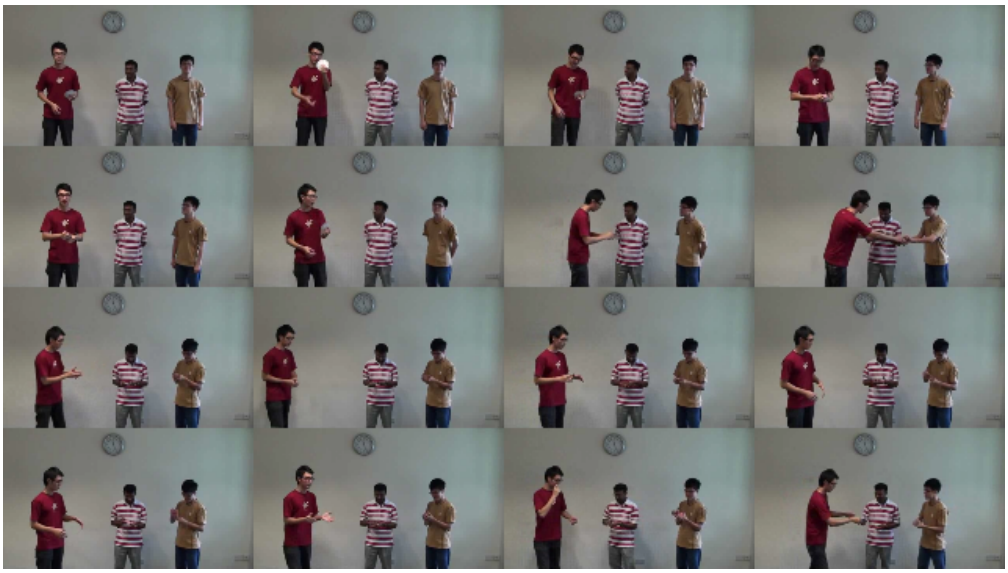


Figure 5.3: Navigation Panel

My solution of this problem is that:

- In function `construction()` I attach function `captureFrame()` to the event `seeked` of the video (line 2 of code 5.3). Thus, each time the video is seeked to a point in time line, the function `captureFrame()` is called.
- In function function `captureFrame()` I capture current frame to the navigation panel and wait 10 milliseconds before seeking the video to next frame (line 20 to 29 of code 5.4). I had tested many times with different videos and found that 10 miliseconds is enough for finishing the function `drawImage()`.

```
1 function construction(){
2     video.addEventListener("seeked", captureFrame);
3     video.controls = false; // disable control video
4     iter = 0;
5     video.currentTime = 0;
6 }
```

Code 5.3: function `construction()`

To prevent unwanted effects when user controls video while the constructing of the navigation panel is running I disable video control (line 3 of code 5.3) before starting the process of navigation panel construction (line 5 of code 5.3).

To help users facilitate follow the process of construction, I show its state (figure 5.4) while the process of construction is running (line 16 to 18 of code 5.4).

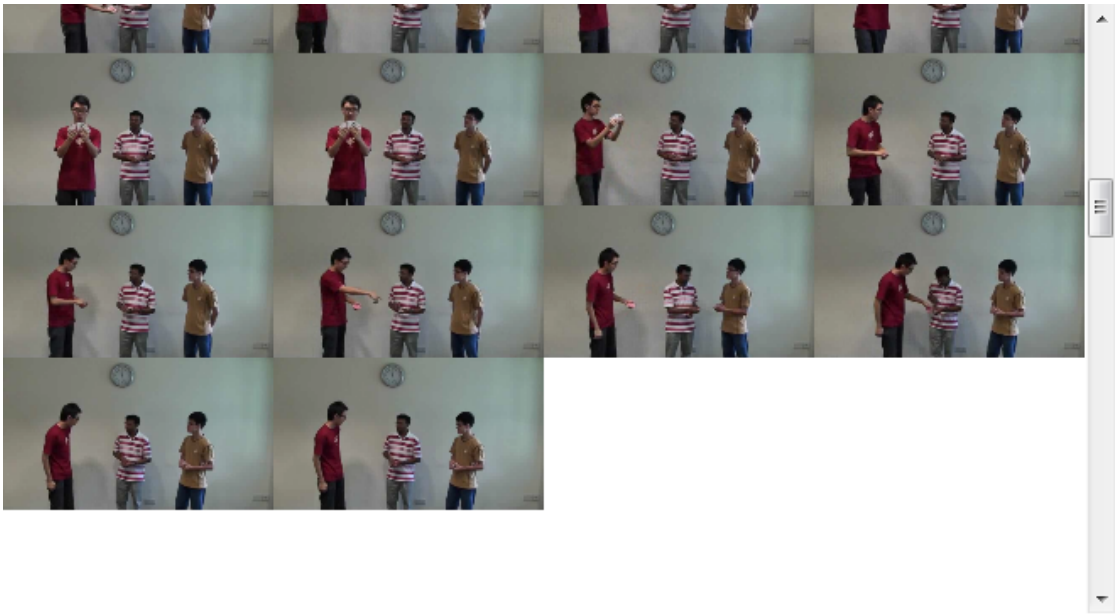


Figure 5.4: State of navigation panel construction

When the process of construction completes, I remove the attachment of function `captureFrame()` on the event `seeked` of video (line 3 of code 5.4) in order to advoid unwanted effects when user plays video. I also enable control video (line 4 of code 5.4) and attach the function `handleClick()` to the event left click on the navigation panel (line 5 to 8 of code 5.4) to make the interactive panel.

```
1  function captureFrame(){
2      if (iter > numFrames){ // keyframe construction completed
3          video.removeEventListener("seeked", captureFrame);
4          video.controls = true; // enable control video
5          // make the interactive panel
6          nav.addEventListener("mousedown", function(event){
7              if(event.button == 0) handleClick(event);
8          }, false);
9
10         video.currentTime = 0;
11         ptitle.innerHTML = "Navigation Panel";
12
13         return;
14     };
15
16     // show process of keyframes construction
17     var state = Math.round((iter*100/numFrames)*100)/100;
18     ptitle.innerHTML = constructing + state + '%';
19
20     // capture frame to canvas
```

```
21      var fX = (iter % navCols) * fWidth;
22      var fY = (Math.floor(iter/navCols)) * fHeight;
23      context.drawImage(video, 0, 0, video.width, video.height, fX, fY,
        fWidth, fHeight);
24
25      // wait 10 milliseconds before seeking to next frame
26      setTimeout(function(){
27          iter++;
28          video.currentTime = iter*intVal/1000;
29      }, 10);
30  }
```

Code 5.4: function `captureFrame()`

Based on the coordinate of the mouse in the panel the function `handleClick()` discovers which frame is clicked and finds its position in the navigation panel, then computes the position which need to jump to in the time line of the video.

```
1  function handleClick(event){
2      var pos = new Array(event.layerX, event.layerY); // position of the
       mouse in the nav
3      var row = Math.floor(pos[1]/fHeight);
4      var col = Math.floor(pos[0]/fWidth);
5      var fpos = row*navCols + col; // position of frame in the nav
6      video.currentTime = fpos*intVal/1000;
7  }
```

Code 5.5: function `handleClick()`

2. Alternative Construction Methods

2.1. Select Keyframe Method

In this section, I devise an alternative method for the construction of navigation panel called *Select Keyframe method*. Unlike in the *Advanced Video Player* (section 1) keyframes are created each 2 seconds, in this method keyframes are created manually by clicking on the button `SELECT FRAME` while user is watching video (figure 5.5).

To implement this construction method, my main idea is that:

- At the begining, I create a global array called `keyframes` to contain selected keyframes.
- Each time the button `SELECT FRAME` is clicked, function `captureFrame()` is called to capture current frame to the navigation panel (figure 5.6). This function creates an object called `frame` which contains position of current frame *in the panel* and *in the timeline* of video, then pushes it to the array `keyframes` and draws the frame in the panel.

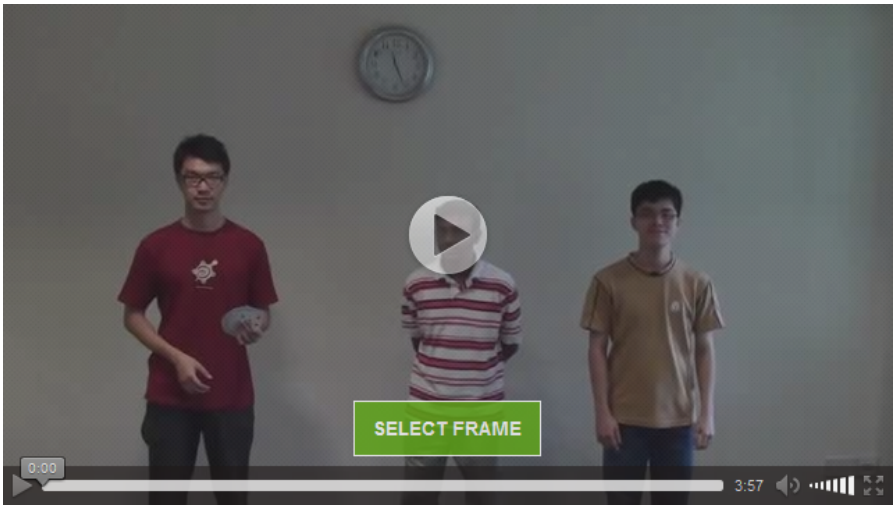


Figure 5.5: UI of Select Keyframe method

```
function captureFrame(){
    var state = isPlaying;
    video.pause();
    video.controls = false; // disable control video
    selFrame.disabled = true; // disable click on the button
    var frame = {pos: keyframes.length, time: video.currentTime};
    keyframes.push(frame);

    var rows = Math.floor(keyframes.length/navCols) + 1;
    if((rows > navRows)&&(keyframes.length%navCols != 0)){ // change
     the number of rows
        navRows = rows;
        nav.height = navRows*fHeight;

        // repaint key frames
        video.addEventListener("seeked", repaint);
        iter = 0;
        video.currentTime = keyframes[iter].time;
    }else{ // do not have change in number of rows
        iter = keyframes.length - 1;
        capture();
        video.controls = true; // enable control video
        selFrame.disabled = false; // enable click on the button
        ptitle.innerHTML = "Navigation Panel";
    }
    if(state){ video.play(); }
}
```

Code 5.6: function `captureFrame()`

- Each time a keyframe in the panel is clicked, function `handleClick()` is called.

This function bases on coordinate of the mouse in the panel to find the position of clicked keyframe in the panel. This is also position of this keyframe in the global array `keyframes`. Based on the time of keyframe is stored in this array, the function `handleClick()` seeks video to the time at which this frame is selected.

```
function handleClick(event){
    if(!video.controls){ return; }
    var pos = new Array(event.layerX, event.layerY); // position of
    the mouse in the nav
    var row = Math.floor(pos[1]/fHeight);
    var col = Math.floor(pos[0]/fWidth);
    var fpos = row*navCols + col; // position of frame in the nav
    if(fpos >= keyframes.length){ return; }
    video.currentTime = keyframes[fpos].time;
}
```

Code 5.7: function `handleClick()`

To advoid unwanted effects, I create some triggers such as: disable control video, disable button `SELECT FRAME`, disable interactive panel... while process of constructing the panel is running. Like the *Advanced Video Player* (section 1), *Select Keyframe method* works well with different videos.

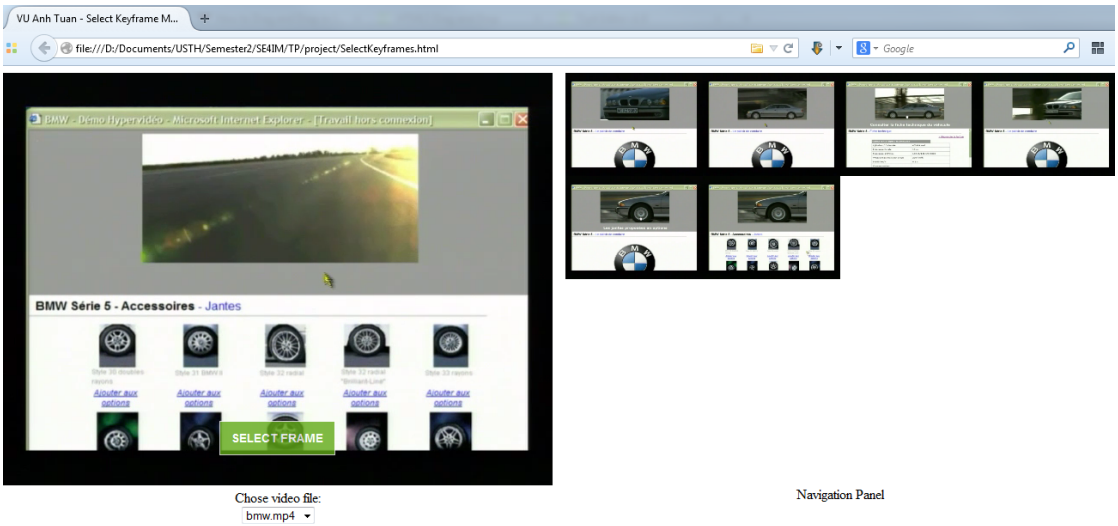


Figure 5.6: Select Keyframe method with the video `bmw.mp4`

2.2. Drag and Drop Method

In this section, I construct another alternative method for the construction of navigation panel called *Drag and Drop method*. In this method, user drags and drops a frame from the video to the navigation panel to create a keyframe.

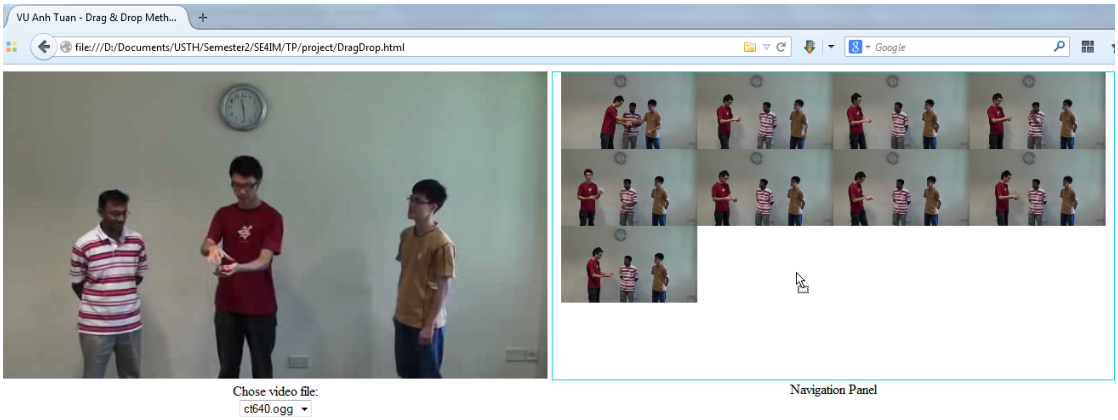


Figure 5.7: UI of Drag and Drop method

The implementation of this method is very similar to the one of *Select Keyframe method* (section 2.1). The biggest different between these methods is that: instead of assigning the function `captureFrame()` to the event `onclick` of button `SELECT FRAME`, in this method I assign the function `captureFrame()` to the event `ondrop` of object `container` which contains the navigation panel.

Likely in the method *Select Keyframe* (section 2.1), in this method I also create some triggers such as disable drag & drop on the video while the construction of navigation is working and enable it when the construction completed... This method also works on different videos.

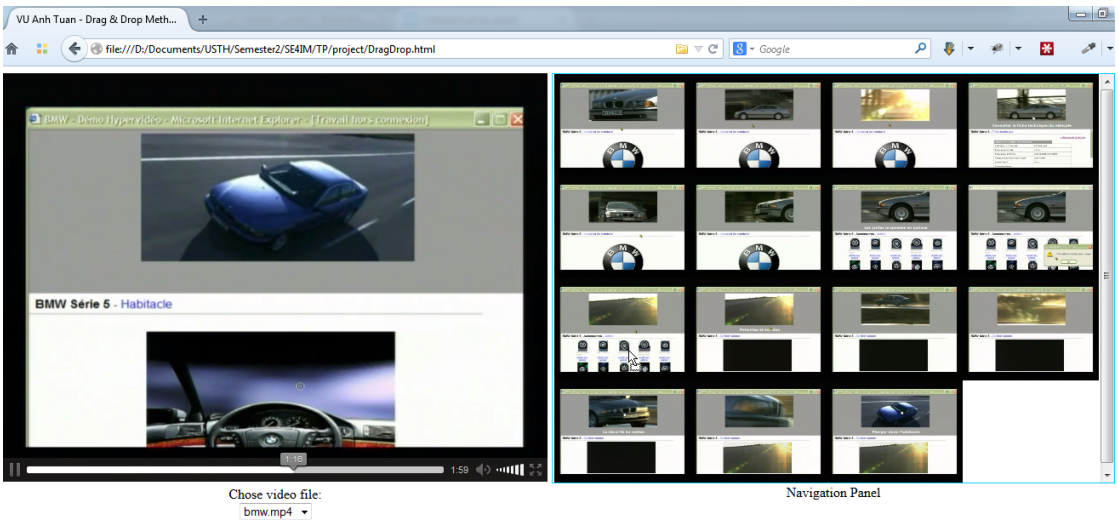


Figure 5.8: Drag and Drop method with the video bmw.mp4

2.3. Content-based Method

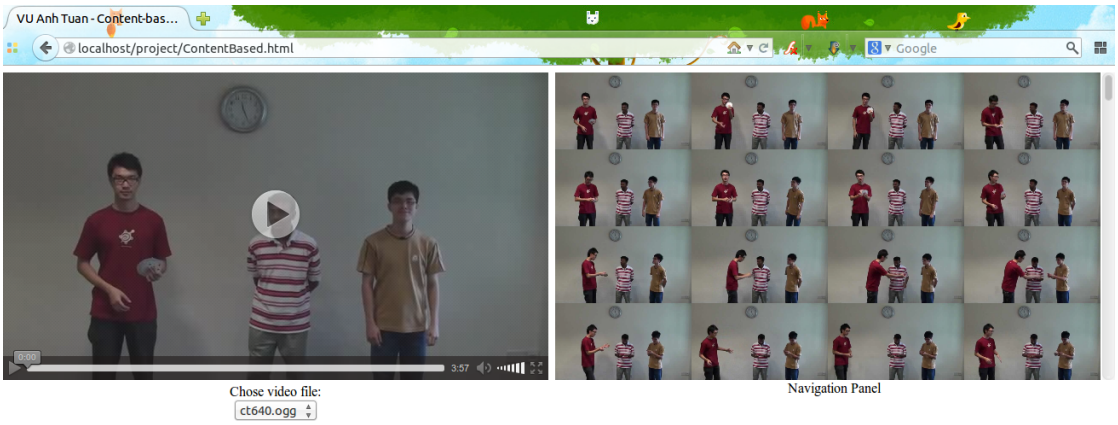


Figure 5.9: UI of Content-based mehod

In this section, I design an interesting method of keyframes construction called *Content-based method*. Unlike previous methods, this method is a content-detected method. In the section 2.1 and 2.2 keyframes are selected manually, in the *Advanced Video Player* (section 1) keyframes are selected automatically each 2 seconds, and in this method a keyframe is chosen if it is significantly different from the previous one.

There are many method to detect the difference between two images such as keypoint matching or using histogram difference [12] or using a library of jquery [13]... However in this section, I use a simple method, *Euclidean distance method* to calculate the difference between two successive frames because this method is simple, it corresponds with the computation in javascript. The second reason is that I compare two frames in a video, so to simplify I suppose that the cases in which the transformation of an frame do not occur. In those cases we need a more complex algorithm to differentiate two frames.

In implementation, this method composes two main steps: frames computation and the construction of navigation panel. At the beginning, the function `initialize()` is called to set default value and compute number of frames in the video.

```
1  function initialize(){
2      video = document.getElementById("video");
3      nav = document.getElementById("nav");
4      ptitle = document.getElementById("ptitle");
5      context = nav.getContext("2d");
6      frameRate = 2; // set frameRate = 2fps
7      intVal = Math.floor(1000/frameRate); // # of milliseconds between
       frame updates
8      navCols = 4;
9      video.width = video.videoWidth;
10     video.height = video.videoHeight;
```

```
11      keyframes = new Array();
12      threshold = 20;
13
14      fWidth = Math.floor(video.width/navCols);
15      fHeight = Math.floor(video.height/navCols);
16      numFrames = Math.floor(video.duration*1000/intVal);
17      nav.width = video.width;
18      nav.height = video.height;
19
20      var container = document.getElementById("container");
21      container.style.width = video.width + 20 + 'px';
22      container.style.height = video.height + 'px';
23
24      construction();
25  }
```

Code 5.8: function `initialize()`

There are two important things in the function `initialize()`: set value on frame rate of video *(line 6 of code 5.8)* and set a threshold to differentiate two frames *(line 12 of code 5.8)*. Since there is no property in html5 which allows to define frame rate of a video, it is also so complex to find an exact value of frame rate of a video in html5 and this is not objective of this section. Therefore to simplify I set value on this variable.

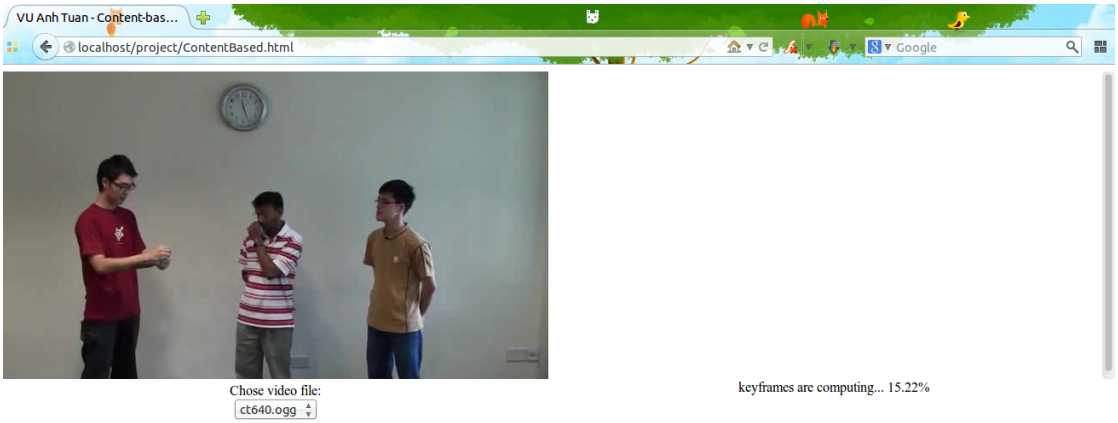


Figure 5.10: Keyframes computation

In the first step function `computeFrame()` is called to take keyframes to a global variable `keyframes`. The main idea of this step is that this function seeks frame by frame in the video and computes the difference between two successive frames. This value is added to a variable called *accumulative difference*. This is the difference between two successive keyframes in the navigation panel. If the accumulative difference is greater than the `threshold`, the second frame is taken as a keyframe in the navigation panel and the accumulate difference is reset to 0 *(line 35-53 in code 5.9)*. Otherwise, the function `computeFrame()` seeks to the next frame and continues update and compare the accumulative difference with the threshold.

```
function computeFrame(){
    var frame;
    var currFrame;
    var hiddenCanvas;
    var hiddenCtx;

    if (iter > numFrames){ // keyframe computation completed
        video.removeEventListener("seeked", computeFrame);

        navRows = Math.ceil(keyframes.length/navCols);
        nav.width = video.width;
        nav.height = navRows*fHeight;

        video.addEventListener("seeked", buildFrame);
        iter = 0;
        ptitle.innerHTML = "Computation complete!";
        video.currentTime = 0;

        return;
    };

    // show process of compute keyframes
    var state = Math.round((iter*100/numFrames)*100)/100;
    ptitle.innerHTML = computing + state + '%';

    // capture frame to canvas
    hiddenCanvas = document.createElement("canvas");
    hiddenCanvas.width = video.width;
    hiddenCanvas.height = video.height;
    hiddenCtx = hiddenCanvas.getContext("2d");
    hiddenCtx.drawImage(video, 0, 0);
    setTimeout(function(){
        iter++;

        if(video.currentTime == 0){
            prevFrame = hiddenCtx.getImageData(0, 0, video.width, video.
    height);
            accumDiff = 0;
            frame = {pos: 0, time: 0, accumDiff: 0, diff: 0};
            keyframes.push(frame);
        } else{
            currFrame = hiddenCtx.getImageData(0, 0, video.width, video.
    height);
            var diff = compareFrames(prevFrame, currFrame);
            accumDiff += diff;
            if(accumDiff >= threshold){
                frame = {pos: keyframes.length, time: video.currentTime,
    accumDiff: accumDiff, diff: diff};
                keyframes.push(frame);
                accumDiff = 0;
            }
            prevFrame = currFrame;
        }
```

```
51
52          video.currentTime = iter*intVal/1000;
53      }, 10);
54  }
```

Code 5.9: function `computeFrame()`

The difference between two frames is computed by the function `compareFrames()`. This function takes Euclidean difference of two frames and normalizes it by the number of pixels as shown in the code 5.10.

```
1  function compareFrames(frm1, frm2){
2      var len = frm1.data.length;
3      var r, g, b;
4      var mean = 0;
5      for(var i=0; i<len-4; i+=4){
6          r = Math.pow(frm1.data[i] - frm2.data[i], 2);
7          g = Math.pow(frm1.data[i+1] - frm2.data[i+1], 2);
8          b = Math.pow(frm1.data[i+2] - frm2.data[i+2], 2);
9          mean += Math.sqrt(r + g + b);
10     }
11     mean = mean/(frm1.width * frm1.height);
12     return Math.round(mean*100)/100;
13 }
```

Code 5.10: function `compareFrames()`

When the first step *frames computation* completes, the function `buildFrame()` is called to perform the second step *construction of navigation panel*. Based on the position and time of each frame in the global variable `keyframes`, this function shows computed keyframes in the navigation panel.

```
1  function buildFrame(){
2      if(iter == keyframes.length){ // build complete
3          video.removeEventListener("seeked", buildFrame);
4
5          video.controls = true; // enable control video
6          // make the interactive panel
7          nav.addEventListener("mousedown", function(event){
8              if(event.button == 0) handleClick(event);
9          }, false);
10
11         video.currentTime = 0;
12         ptitle.innerHTML = "Navigation Panel";
13
14         return;
15     }
16
17     // show process of compute keyframes
18     var state = Math.round((iter*100/keyframes.length)*100)/100;
19     ptitle.innerHTML = constructing + state + '%';
```

```
20
21      var fX = (iter % navCols) * fWidth;
22      var fY = (Math.floor(iter/navCols)) * fHeight;
23      context.drawImage(video, 0, 0, video.width, video.height, fX, fY,
        fWidth, fHeight);
24
25      // wait 10 milliseconds before seeking to next frame
26      setTimeout(function(){
27          iter++;
28          if(iter < keyframes.length){
29              video.currentTime = keyframes[iter].time;
30          } else{
31              buildFrame();
32          }
33      }, 10);
34  }
```

Code 5.11: function `buildFrame()`

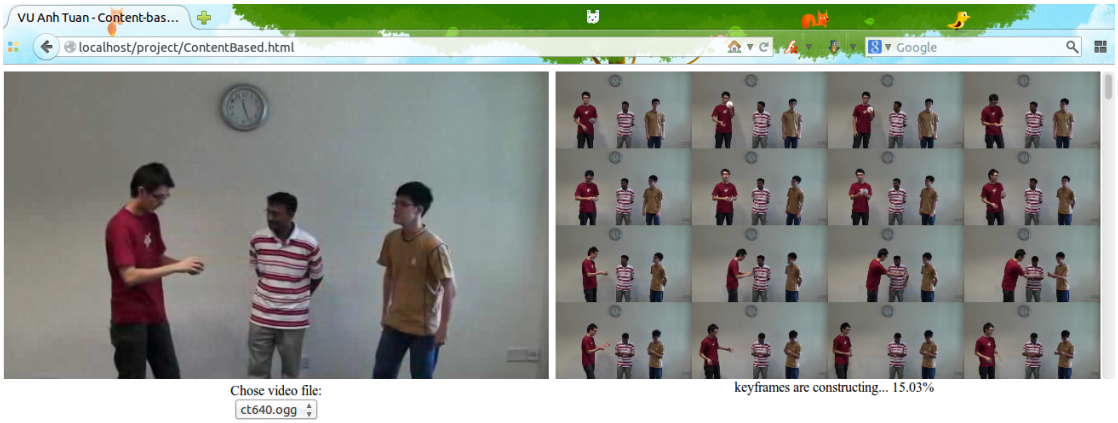


Figure 5.11: Construction of navigation panel

I have tested this method with the video `ct640.ogg` with the `framerate` is set to the real framerate of this video, 25fps and the `threshold` equals 20. The result shows that this method found 432 keyframes over 5000 frames of entire video. This result is quite good because if a user has not the time to watch all video, he/she can watch only 432 keyframes in the navigation panel to understand the content of the video. He/she saves 90% of watching time in comparison with watching all 5000 frames of the video. With the default $framerate = 2$, in my computation, user can save about 75% of watching time if he/she watches only keyframes in the navigation panel instead of watching entire video.

Last but not least since in this method I used function `getImageData()` to obtain RGB value of pixels in a frame, the two execution files html and js need to put on a server to advoid security warning of Firefox browser. I recommand an Apache server.

3. Alternative Navigation Method

3.1. Slider Method

In this section, I devise an alternative method for the navigation panel called *Slider method*. In this method, the navigation panel is presented as many pages of size 1 x 4. At two sides of this panel, there are two buttons to go to the next page and go back to the previous page (figure 5.12).

The implementation of this method is very similar to the one of the Advanced Video Player (section 1). The biggest different between them is that: In this method I change the size of panel to 1x4 instead of 4x4, and create a horizontal scroll instead of vertical scroll in the panel. Then, I create two buttons `prev, next` and create functions to go back and forward for them.

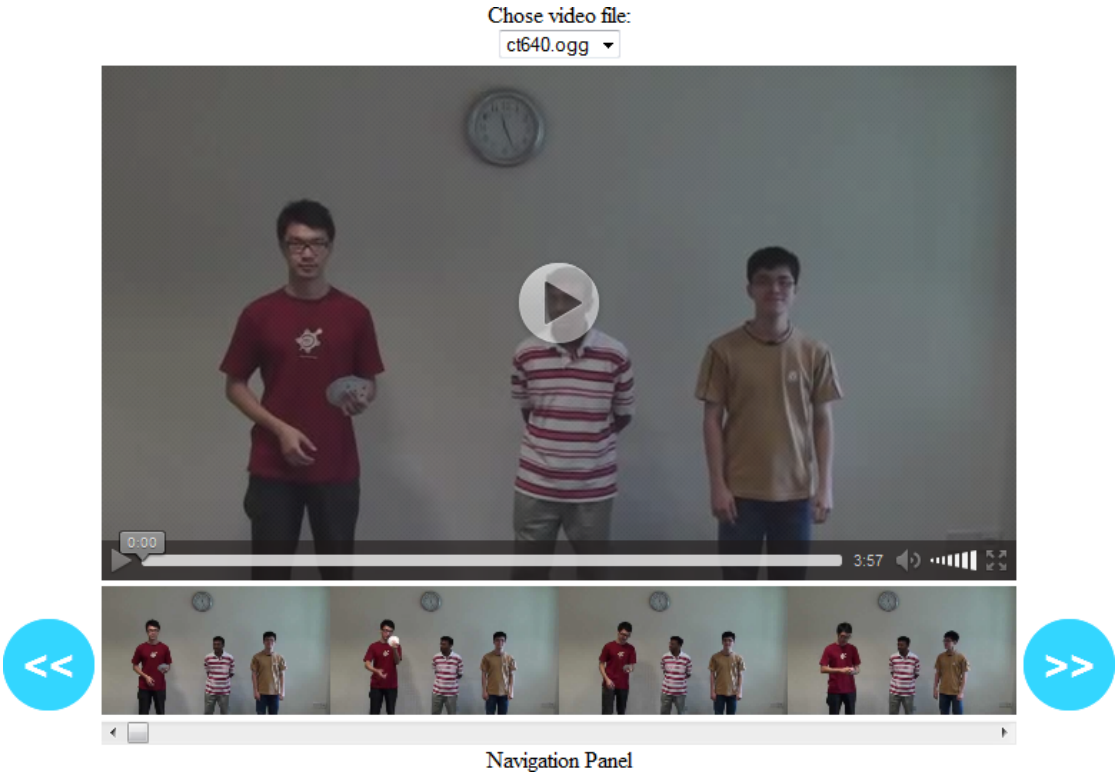


Figure 5.12: UI of Slider method

This navigation method can apply to all construction methods (section 2). However, for simplification I devise this method with the basic construction in Advanced Video Player (section 1).

3.2. Multiresolution Method

In this section I implement another method for the navigation panel called *Multiresolution method*. In this method, when user select a keyframe on the navigation panel, a panel called *Multiresolution panel* is shown to help user choose a fine keyframe to start (figure 5.13). The multiresolution panel has size 3x3 and takes the selected keyframe in the navigation panel into the frame center as shown in the figure 5.13. Frames before centre frame in the multiresolution panel are frames which are between selected keyframe and its previous keyframe in the navigation panel. Frames after centre frame in the multiresolution panel are frames which are between selected keyframe and its after keyframe in the navigation panel.

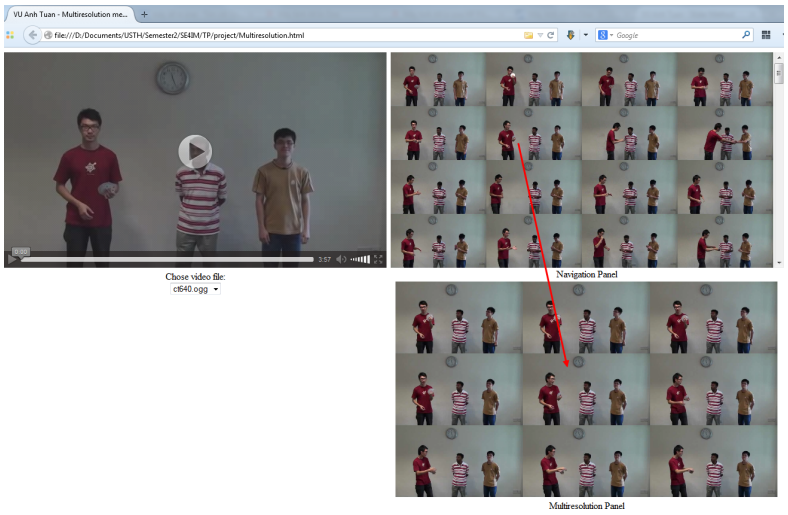


Figure 5.13: UI of Multiresolution method

The implementation of this method compose two main steps: *the construction of navigation panel* and *the construction of multiresolution panel*. The first step *construction of navigation panel* (figure 5.14) is very similar the one of the Advance Video Player (section 1). When this step finish, I attach the function `handleNavClick()` to builds the multiresolution panel (figure 5.15) when user clicks on the navigation panel.

```
1  function handleNavClick(event){
2      var pos = new Array(event.layerX, event.layerY); // position of the
       mouse in the nav
3      var row = Math.floor((pos[1])/fHeight);
4      var col = Math.floor((pos[0])/fWidth);
5      ftime = (row*navCols + col)*intVal/1000; // time of frame in the nav
6
7      // build multiresolution panel
8      vstate = isPlaying;
9      ctime = video.currentTime;
10     video.pause();
11     video.addEventListener("seeked", captureFrameMulRes);
```

```
12      video.controls = false; // disable control video
13      iter = -4;
14      numFrames = 4;
15      context = multires.getContext("2d");
16      context.clearRect(0, 0, video.width, video.height);
17      var seekTo = ftime + iter*mulIntVal/1000;
18      seekTo = (seekTo < 0) ? 0 : (seekTo > video.duration) ? video.
        duration : seekTo;
19      video.currentTime = seekTo;
20  }
```

Code 5.12: function `handleNavClick()`

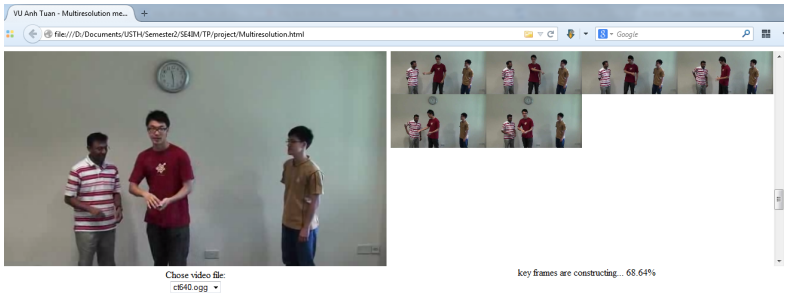


Figure 5.14: The construction of navigation panel

When user click on a keyframe in the navigation panel, based on the position of this keyframe, the function `handleNavClick()` calls the function `captureFrameMulRes()` to build 9 frames around this keyframe as shown in the multiresolution panel of the figure 5.13.

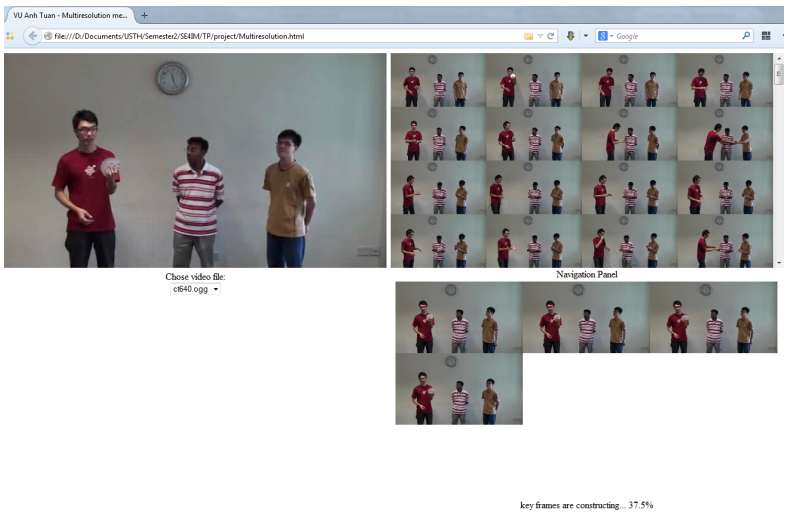


Figure 5.15: The construction of multiresolution panel

```
function captureFrameMulRes(){
    if (iter > numFrames){ // keyframe construction completed
        video.removeEventListener("seeked", captureFrameMulRes);
        video.controls = true; // enable control video
        // make the interactive multiresolution panel
        multires.addEventListener("mousedown", function(event){
            if(event.button == 0) handleClick(event);
        }, false);

        video.currentTime = ctime;
        mtitle.innerHTML = "Multiresolution Panel";
        if(vstate) video.play();
        return;
    };

    // show process of keyframes construction
    var state = Math.round(((iter+4)*100/(numFrames+4))*100)/100;
    mtitle.hidden = false;
    mtitle.innerHTML = constructing + state + '%';

    // capture frame to canvas
    var fX = ((iter+4) % mulCols) * fWidthMul;
    var fY = (Math.floor((iter+4)/mulCols)) * fHeightMul;
    context.drawImage(video, 0, 0, video.width, video.height, fX, fY,
    fWidthMul, fHeightMul);

    // wait 10 milliseconds before seeking to next frame
    setTimeout(function(){
        iter++;
        var seekTo = ftime + iter*mulIntVal/1000;
        seekTo = (seekTo < 0) ? 0 : (seekTo > video.duration) ? video.
    duration : seekTo;
        video.currentTime = seekTo;
    }, 10);
}
```

Code 5.13: function `captureFrameMulRes()`

4. Conclusion

This is a very interesting project. In this project I had construct an Advance Video Player based on the knowledge of HTML5 and Javascript. I also improved a little this video player by giving different methods to construct the navigation panel such as *Select Keyframe method, Drag and Drop method* and *Content-based method.*

Among these method I am really impressed by *Content-based method.* It extracts automatically keyframes based on the different between frames in video. The process of doing this method also helps me to understand the ways they compute the difference of two images in general and of two frames in a video in particular.

Last but not least, the process of doing the alternative navigation methods help me to have a profound understanding of interaction between a system with users.

The project also gives me some open questions such as how to compute the framerate of a video in HTML5, what is the best value of threshold to detect keyframes with different types of video... In the future, if I have a chance I will try to answer these questions.

Glossary

1. *Media*: any form of information, including text, audio, two- and three- dimensional graphics, animation, photo images, videos, etc.
2. *Transmission*: the transfer of data over a communications channel.
3. *Compression*: is used to reduce the size of one or more files. While file compression is performed using a lossless compression algorithm, most image, audio and video files are compressed using lossy compression algorithm.
4. *Interaction*: a process by which two or more things affect each other.
5. *Content*: any information that available for retrieval by the user, including web pages, images, music, audio, etc.
6. *User*: anyone who requires the use of services of a computing system or its products.
7. *Synchronisation*: keeping the clocks of two things (i.e. videos, devices, etc.) beating at the same rate.
8. *Protection*: the action of applying measures to maintain and sustain the integrity of existing materials (i.e. text, videos, images, etc.).
9. *Streaming*: method of transmitting a media file in a continuous stream of data that can be processed by the receiving computer before the entire file has been completely sent.
10. *Social media*: the online forms of communicating that any individual can employ, which include blogs, microblogs such as Twitter and social networking sites such as Facebook.
11. *Tag*: a unit of information used as a label or marker.
12. *Semantic*: the study of the meaning of words and phrases.
13. *Hypervideo*: is a displayed video stream that contains embedded, user-clickable anchors, allowing navigation between video and other hypermedia elements.
14. *3D models*: represent a 3D object using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, etc.

15. *Story*: a conversation that is taking place at different time and places during a project between the various stakeholders concerned by the given feature (customers, users, developers, testers, etc.) which is largely verbal but most often supplemented by documentation.

Bibliography

[1] Vincent CHARVILLAT. *Course Software Engineering for Interactive Media*. University of Science and Technology of Hanoi (USTH), March/April, 2014.

[2] Giannis Tsakonas. *User studies - Enquiry foundations and methodological considerations*. 1^{st} Workshop on Digital Information Management, March, 2011.

[3] R. Kosara, C.G. Healey, V. Interrante, D. H. Laidlaw, and C. Ware. *Thoughts on Users Studies: Why, How and When*. IEEE Computer Graphics & Applications (CG&A), Visualization Viewpoints, vol. 23, no. 4, pp. 20–25, 2003.

[4] Wei Tsang Ooi *Javascript and HTML5 Canvas*. National University of Singapore, Course Networked and Mobile Gaming, 2012/2013.

[5] *HTML5 Canvas*. W3schools.com.

[6] *Function.apply and Function.call in JavaScript*. Ode to Code, July 2007.

[7] Kai-Yin Cheng, Sheng-Jie, Bing-Yu, and Hao-Hua Chu. *SmartPlayer: User-Centric Video Fast-Forwarding*. ACM CHI 2009 Conference Proceedings, pp. 789–798, 2009.

[8] Jcinwa. *TiVo Reviews*. Product Review, Feb, 2014.

[9] Xueliang Liu, Tao Mei, Xian-Sheng Hua, Bo Yang, and He-Qin Zhou. *Video collage*. Proceeding MULTIMEDIA '07 Proceedings of the 15th international conference on Multimedia, pp. 461-462, 2007.

[10] Michael Smith and Takeo Kanade. *Automated Video Skimming*. Carnegie Mellon University, Informedia Digital Video Library Project.

[11] Peter Lubbers, Brian Albers and Frank Salim. *Pro HTML5 Programming, Chapter 3: Working with HTML5 Audio and Video*. Apress, August, 2010.

[12] Ganesh I. Rathod, Dipali A. Nikam. *An Algorithm for Shot Boundary Detection and Key Frame Extraction Using Histogram Difference*. International Journal of Emerging Technology and Advanced Engineering, ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 8, August 2013.

[13] James Cryer and the Huddle development team. *Resemble.js : Image analysis and comparison*. Github, Feb, 2013.